Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Algorithms Performance and Reliability Assessment

GORDA Deliverable D5.1

Due date of deliverable: 2006/09/30
Actual submission date: 2006/09/30
Revision date: 2007/04/12

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

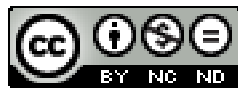
Universidade do Minho

Revision 1.2

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

Contributors

Alfrânio Correia Júnior
José Pereira
Luís Soares
Nuno Alexandre Carvalho
Rui Oliveira



(C) 2006 GORDA Consortium. Some rights reserved.
This work is licensed under the Attribution-NonCommercial-NoDerivs 2.5 Creative Commons License.
See http://creativecommons.org/licenses/by-nc-nd/2.5/legalcode for details.

Contents

1 Introduction **6**
1.1 Problem Statement 6
1.2 Contributions 7
1.3 Outline 7

2 Evaluation Platform **9**
2.1 Background 9
2.1.1 Testing Environments and Benchmarks 9
2.1.2 Simulation 10
2.1.3 Centralized Simulation 11
2.2 System Architecture 14
2.3 Simulation Model 15
2.3.1 Workload Model 15
2.3.2 Transaction Processing Model 16
2.3.3 System Under Test 18
2.3.4 Network Model 19
2.4 Simulation Kernel 19
2.4.1 Real-Time 19
2.4.2 Client/Server Utility Classes 20
2.4.3 Library 22
2.5 Model Calibration 22
2.5.1 CPU 23
2.5.2 Storage 23
2.6 Validation 25

3 Protocol Evaluation **28**
3.1 Motivation 28
3.2 Scenarios 30
3.3 Results: Optimistic vs Conservative 31
3.3.1 Coarse Grain 31
3.3.2 Fine Grain 33
3.3.3 Snapshot Isolation 33
3.4 Results: Wide-area Inter-Cluster Enhancements - WICE 34

3.4.1 Performance . . . 35
3.4.2 Discussion . . . 37
3.5 Results: Fault-injection and DBSM . . . 37
3.5.1 Termination protocol performance . . . 37
3.5.2 Performability: Graceful Degradation . . . 39

4 Conclusion **42**

A Distribution Parameter Estimation **44**

B Real vs Simulation Model **46**

C Usage Examples **48**
C.1 Start . . . 48
C.2 DML . . . 48
C.3 Simple Network Hello Application . . . 49
C.4 Simple Timer Application . . . 51
C.5 Locks Example . . . 52

List of Figures

2.1 Real implementation: Client/Server interaction UML diagram. 11
2.2 Simulation: Client/Server interaction UML diagram. 12
2.3 Executing simulated and real jobs. 12
2.4 Scheduling events from real code. 13
2.5 Simple overview of the system architecture. 14
2.6 Simulated transactions and replicas interaction. 17
2.7 Simulated client calls server real implementation. 20
2.8 Client real implementation calls simulated server). 21
2.9 *XLogWrite* calls as a function of the number of items accessed. 24
2.10 Validation of the centralized simulation runtime. 26
2.11 Simulation vs Real comparison. 27

3.1 Local area network configuration. 30
3.2 Wide area network configuration. 31
3.3 Performance measurements in a LAN with coarse granularity. 32
3.4 Detailed profiling in a LAN with 270 clients. 33
3.5 Performance measurements in a LAN with fine granularity. 34
3.6 Performance measurements in a LAN with snapshot isolation. 35
3.7 Performance results with 1-SI. 36
3.8 Certification latency (fault injection). 39
3.9 P($L \leq \tau$). 40
3.10 Performance comparison. 41

A.1 Processing time fitting. 45

B.1 Transaction performance comparison. 47

List of Tables

2.1 Transaction types in TPC-C. 15
2.2 Size of tables in TPC-C. 16
2.3 System specifications. 22
2.4 Transaction CPU times distribution and estimated parameters (nanosecond precision). . . 24
2.5 Storage simulation model parameters. 25
2.6 Simulation vs Real: TPM and latency results. 26

3.1 Protocol naming and characterization details. 29
3.2 Definition of coarse conflict classes for each transaction type in TPC-C. 31
3.3 Types of faults injected. 38
3.4 Protocol CPU usage (%). 38
3.5 Abort rates with 3 sites and 1000 clients (%). 39

Chapter 1

Introduction

1.1 Problem Statement

This report aims at evaluating the impact of key design decisions on the performance and reliability of database replication protocols being considered in the GORDA project. Namely, we are interested in the detailed evaluation of concurrency control mechanims and in the interaction with group communication protocols with realistic workloads and faultloads.

However, current methodologies of evaluation of replication protocols reveal themselves limited. The existent studies report to analysis conducted in local area networks or in small scale scenarios. The fact is that there are few chances to put together a wide area testing environment because that is not cost-effective, does not provide full control over the environment variables as it does not enable deterministic testing. Furthermore, although in local area environments there is more control over the environment, there is still no deterministic execution and, as in the wide area, a full deployment of the software is also required. These issues must be addressed when researching, developing, evaluating distributed replication protocols. Realistic and controlled performance evaluation of distributed and replicated applications is still very hard to achieve, specially if one considers large scale scenarios and fault injection. Although there are few testbeds [7, 24, 47] that enable the evaluation of distributed systems, they all focus a small portion of the spectrum of possible problems. On the other hand, pure simulation models may be a solution to this problem, but creating abstract models from real implementations is not always possible or even desirable.

The problem extends itself when there is the need for testing early and often during the development cycle of a new replication protocol. The researcher/developer finds himself without the possibility to abstract the irrelevant parts of the system, focussing on the protocol and still conduct a realistic evaluation. For instance, when evaluating a database replication protocol, the database engine and the network could be abstracted as realistic simulation models, since they are not really part of the problem, and the protocol should be the only real implementation. This methodology is known as incremental development. In this context, the development process is backed by an abstract simulation model comprised of multiple sub-models that get replaced by real implementations as these become available. The problem here is having the simulation execution to play nicely with the real execution. Whenever real code execution takes place, the time spent on it must be accounted in the simulation time line. Doing this in an *ad hoc* fashion is time consuming and error prone. Hence the approach of centralized simulation proposed in CESIUM [11], must be taken into account. One cannot reuse CESIUM implementation because it defines specific models that are not completely suitable for evaluating database replication protocols.

1.2 Contributions

Briefly, this report presents replication algorithms evaluation, supported by a simulation framework that enables realistic component modeling. In detail, it combines abstract simulation models and real implementations in a centralized environment [11]. It takes advantage of a centralized execution environment which provides means to embed implementations of protocols into simulation. The framework allows changing the scenarios without having to rewrite any source code, by requiring only modification to configuration files. The evaluation presented is supported by this framework, in which three group communication replication protocols are profiled. The study brings forward conflict related issues and provides a solution as well as evaluates the protocols in local and wide area networks. It also assess the performance degradation when there is fault-injection.

List of contributions:

SSF Extensions for Centralized Simulation
A centralized simulation framework which is based on the SSF specification. It provides transparent means for injecting real implementations prototypes into simulation and synchronize simulation clock with real execution.

SSF-based Database Simulation Model
A database simulation library written in Java. It provides several simulation models mimicking the execution of a real database. Calibration and validation conducted ensures coherent behavior when compared to a real system;

Protocol Evaluation
The evaluation allowed studying the behavior of the optimistic and conservative protocols under different conditions. These ranged from relaxed to strong correctness criteria processes, local area to wide area networks and from faultless to faulty environments.

1.3 Outline

This document structured as follows: Chapter 2 briefly reviews the state-of-the-art in performance and reliability evaluation and then discusses the implementation of a simulation framework and its application to the study of database replication protocols; Chapter 3 evaluates key components of database replication protocols under consideration in the GORDA project by applying the proposed centralized simulation framework; Finally, Chapter 4 concludes the report.

Publications

Portion of the work presented in this report have been previoulsy published in the form of a workshop and conference papers as well as thesis:

- A. Sousa and J. Pereira and L. Soares and A. Correia Jr. and L. Rocha and R. Oliveira and F. Moura. Testing the dependability and performance of GCS-based database replication protocols. In *Proceedings of The International Conference on Dependable Systems and Networks*. 2005 (DSN'05).
- A. Correia Jr. and A. Sousa and L. Soares and J. Pereira and R. Oliveira and F. Moura. Group-based replication of on-line transaction processing servers. In *Dependable Computing: Second*

Latin-American Symposium. 2005 (LADC'05).

- L. Soares and J. Pereira. Experimental performability evaluation of middleware for large-scale distributed systems. In *7th International Workshop on Performability Modeling of Computer and Communications Systems*. 2005 (PMCCS'05).
- L. Soares. Evaluation of Group-based Database Replication Using Centralized Simulation. Masters Thesis. University of Minho, 2006.
- N. Carvalho. MinhaLib: Network and Concurrency Simulation. Diploma Thesis. University of Minho, to appear.
- J. Grov, L. Soares, A. Correia Jr., J. Pereira, R. Oliveira, and F. Pedone. A Pragmatic Protocol for Database Replication in Interconnected Clusters. IEEE Intl. Symp. Pacific Rim Dependable Computing (PRDC'06), 2006.

Chapter 2

Evaluation Platform

2.1 Background

Detailed evaluation of the performance and dependability of abstract protocols and of their implementations is required to examine each trade-off in different environments. Realistic tests are however costly to setup and run and depend on the availability of representative workloads and fault-loads. This is especially difficult when targeting large clusters or wide-area systems. Although often used, toy applications and micro-benchmarks are unable to disclose the subtle interactions with application semantics and dynamics (e.g. flow control issues and hot-spots) and with the environment (e.g. fault scenarios) as well as introduce significant probe effect.

System evaluation also depends on the availability of the complete target system. This precludes incremental development and early testing of individual components. Agile development methodologies have been attracting software engineers to focus on modeling and automated testing of compliance. Unit testing means that development starts by producing auxiliary test components directly from the model which are used, as the software product evolves, to ensure that it matches initial modeling.

2.1.1 Testing Environments and Benchmarks

Benchmarks are used to assess the performance and scalability of IT systems. Stress-driven or specific context driven tests are conducted when running the benchmark. This allows the detection of bottlenecks as well as assessing its validity under not so usual situations, which normally are disregarded during development. Benchmarking a system, involves deploying a suite of applications that setup the testing scenario according to the benchmark specifications. This is actually deploying a fully working system. For instance, if one wants to use TPC-C [46] or SPEC-Web [31] benchmarks, a set of applications and a database needs to be installed. Nonetheless, the overhead of setting up the benchmark, either in terms of resources as in terms of time, may not meet the user requirements.

There are a number of tools to setup and control distributed tests and benchmarks, such as NetBed [49], ACME [7], and TestZilla [47], targeted specifically at performance. A large share of the complexity of such tools is directly related with the distributed nature of the system under study, namely, in performing consistent global observation of system state and properties while minimizing interference. Tools such as Facilita Forecast [24] add to a distributed testing scenario the generation of representative loads, enabling load and stress testing to a wide range of applications. Distributed testing can also be performed in realistic testbeds for wide area networks such as Emulab [2] and PlanetLab [5] which provide trans-

parent usage of the resources. They provide flexibility but the user always need to deploy a real system to perfrom the test.

2.1.2 Simulation

Simulation is often defined as the technique to mimic behavioral interactions of a given environment, using a specified abstract model or instrument. The model or instrument are used to obtain reliable and valuable information on how the real system should evolve through time or even as a training playground [48]. Models may be categorized into physical or mathematical.

When modeling a system, one must take into account that the model should not represent the system *per se*. In fact, it should be a simplification of the real system. Nevertheless, the aspects under study should hold sufficient detail to draw valid conclusions about the real system [12].

Simulation of computer systems uses the discrete event approach: Time advances as events are scheduled to happen. Therefore, if the system is just leaving instant in time t and the next event is scheduled to happen at instant t', the simulation timeline jumps from t to t' instantaneously. In such approach, components trigger events which are scheduled to happen in the future creating a dynamic simulation time flow. Components tend to be defined at a considerable level of abstraction, hence designing large scale simulation models is acceptable as well as their computation time. A nice example of how to use this simulation strategy is a CPU model. It may be modeled as a simple queue in which events are defined as the: the arrival of a job and the departing of a job. In detail, upon a job, J_0, arrival, it is set to execute at the CPU. If a job, J_1, arrives in the mean time, it is put on hold. The service time is simulated by scheduling an event to happen at the end of the job execution. When the execution reaches the end, J_0 finishing the event is triggered, and job J_1 is set to execute in the CPU. The work presented in this report considers the discrete-event strategy to operate the simulation timeline, hence the continuous strategy will not be addressed from this point on.

An implementation of a discrete event simulation approach is the Scalable Simulation Framework (SSF) [19]. To build an SSF model, one identifies the objects of interest, *entities*, and their *attributes*. Entities may have their attributes changed over the time by *processes*, also named *activities*, which are triggered by the occurrence of *events*. Moreover, the system maintains a *state*, the collection of variables which the modeler specifies as being interesting for the study, describing the system at any time.

The SSF interface specifies additionally *incoming* and *outgoing channels* which act as event routes between entities and their associated processes. Delays are imposed on individual events, but also on channels and channel bindings, allowing the system to partition the simulation and take advantage of parallel processing [32, 30, 33]. The specification includes also the Domain Modeling Language (DML) [1] that can be used to assemble models. Multiple implementations of the SSF interface are available in C++ [32, 3] and Java [6]. The simple and clear interface of SSF provides means for intuitively and easily setting up simulation models. As an example, if one wants to simulate message passing in a network, one may bind channels between entities, acting as network cards, in a point-to-point manner. Events written into channels represent messages. Messages get delayed in the "network" accordingly to the delay specified at channel binding time or may be imposed specifically for each message when the event write operation takes place. These kind of setups may be configured using only a simple DML configuration.

A more detailed example is shown in Figure 2.1 and Figure 2.2. These compare real and simulated client/server interaction, respectively. Note that in these diagrams, and for the sake of comprehensibility, channels and entities are implicit, hence, asynchronous events suffixed with *Event* translate into write and reads into implicit channels. In Figure 2.1, the time it takes to execute a request is the sum of the partial latencies of calling, executing and receiving back the return value. The identical scenario is depicted in

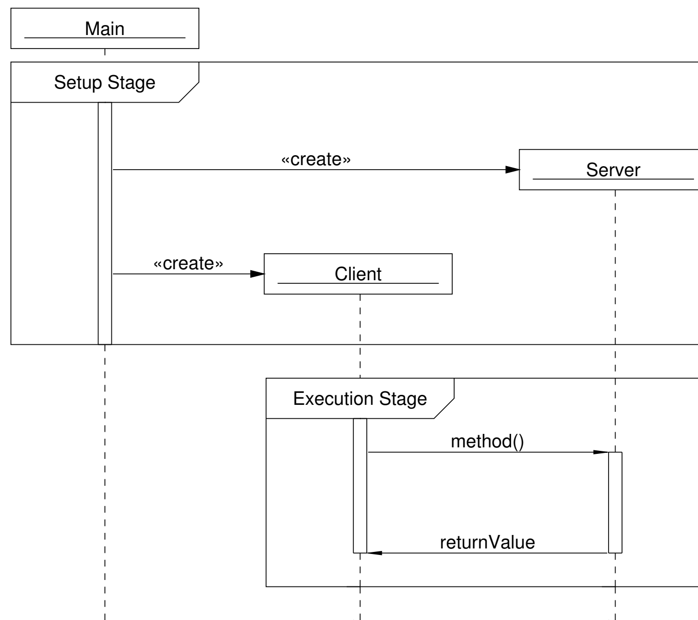


Figure 2.1: Real implementation: Client/Server interaction UML diagram.

Figure 2.2, but this time using pure simulation. In this case, direct invocations are translated into events and simulation models are used instead of real implementations. This means that both, client and server, are now entities with associated processes. These are connected using channels from and on which events are written and read. As in the real implementation, the execution latency is determined by the time spent calling the method, executing the request and getting the return value. But now, invocation, execution, and reply latencies are modeled as event delays. Therefore, one may say that the simulation clock advances horizontally, each time an event operation (read/write to a channel is performed) and real time advances vertically as native instructions get executed.

A popular approach to evaluate designs, and specifically in the study of the dynamic properties of very large and complex systems, is the development of simulation models. Namely, the development of the network infrastructure, protocols, and their applications is based on tools such as ns-2 [4] and SSFNet [20]. Although simulation models are usually distinct from final implementations, often using scripting languages to simplify modeling, it is sometimes possible to reuse library code by wrapping the functionality of simulated components in standard APIs, thus allowing functional testing.

Fine grained simulation of computer systems can also be used to create highly realistic although small scale testbeds. Namely, tools such as Simics and SimOS [41] simulate in detail computer systems allowing the execution of COTS binary-only operating systems and application software with unparalleled observability and lack of interference. On the other hand, the detail means that substantial computing resources are required to run realistic loads and that full implementations are required for testing. This can be improved by directly running operating system and application code in the host processor. This is the approach of UMLSim [10] and FAUMachine [16], which additionally allow for fault injection for dependability evaluation. Nevertheless, full implementations are still required for testing.

2.1.3 Centralized Simulation

The ability to evaluate real implementations in a simulated environment has been proposed in CESIUM [11]. In this framework, implementations of communication protocols are tested for real-time

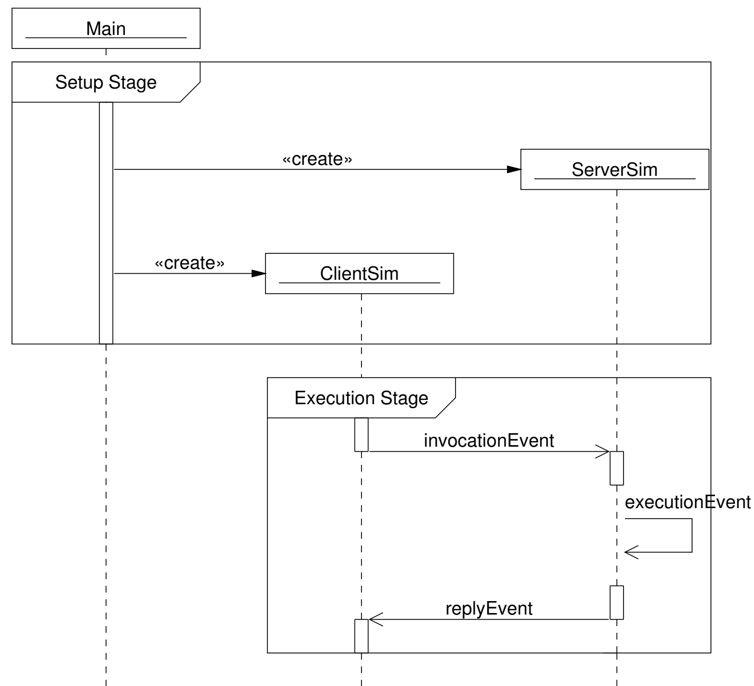


Figure 2.2: Simulation: Client/Server interaction UML diagram.

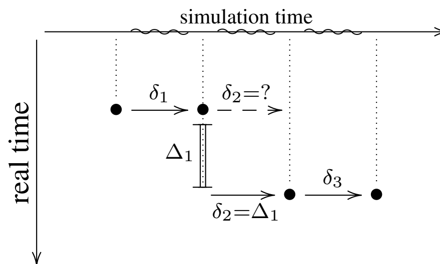


Figure 2.3: Executing simulated and real jobs.

properties. By running multiple instances of the implementation in a single address space within a discrete-event simulation model of the environment, centralized observation and manipulation of state is allowed with reduced interference. Scheduling and management of virtual-time by taking account of real time consumed by implementations, as obtained by profiling them, allows the system to be tuned to accurately reproduce real systems. This approach is thus the only that allows incremental substitution of model components by real implementations in performance evaluation.

In an event-driven simulation, time is incremented only by scheduling events with non-zero delays. The challenge when mixing real and simulated components is to ensure that the time actually spent executing real code is accurately reflected in simulation time. Thus this approach goes beyond the simple reuse of real code for simulation models and is able to reproduce timing properties of real systems [11].

In detail, this implies starting a profiling timer whenever real code is entered to account for native execution time. When execution re-enters simulation code, the profiling timer is stopped and the elapsed time used as an offset for all events scheduled. This is illustrated in Figure 2.3. From the simulation point of view, a job with duration δ can be set to execute at a specific instant t by scheduling a simulation event to enqueue it at simulated time t. Executing jobs with real code is layered on top of the same simulation

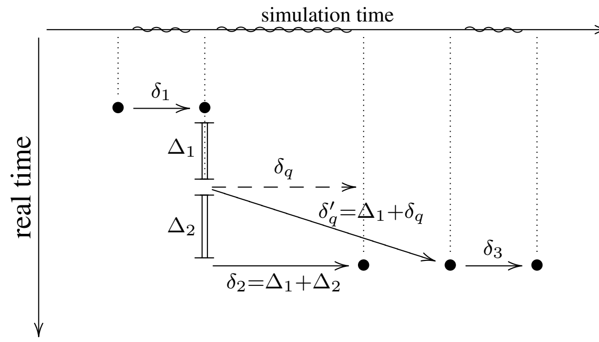


Figure 2.4: Scheduling events from real code.

mechanism. Figure 2.3 illustrates this with an example of how three queued jobs are executed. The second job is assumed to contain real code. The x-axis depicts simulated time and the y-axis depicts relevant real-time (i.e. we ignore real-time consumed during execution of pure simulation code and thus pure simulation progresses horizontally). The x-axis shows also with an wiggly line when the simulated execution is taking place. Solid dots represent the execution of discrete simulation events. Scheduling of events is depicted as an arrow and execution of real code as a double line.

The first job in the queue is a simulated job with duration δ_1. After δ_1 has elapsed, execution proceeds to a real job. In contrast with a simulated job, one does not know beforehand which is the duration δ_2 to be assigned to this job. Instead, a profiling timer is started and the real code is run. When it terminates, the elapsed time Δ_1 is measured. Then $\delta_2 = \Delta_1$ is used to schedule a simulation event to proceed to the next job. This brings into the simulation timeline the elapsed time spent in a real computation. Finally the second simulated job is run with duration δ_3.

As a consequence of such setup, queuing (real code or simulated) jobs from simulated jobs poses no problem. Only when being run, they have to be recognized and treated accordingly. Problems arise only when real code needs to schedule simulation events, for instance, to enqueue jobs at a later time. Consider in Figure 2.4 a modification of the previous example in which the third job is queued by the real code with a delay δ_q. If real code is allowed to call directly into the simulation runtime two problems would occur:

- Current simulation time still doesn't account for Δ_1 and thus the event would be scheduled too early. Actually, if $\delta_q < \Delta_1$ the event would be scheduled in the simulation past!
- The final elapsed real time would include the time spent in simulation code scheduling the event, thus introducing an arbitrary overhead in δ_2.

These problems can be avoided by stopping the real-time clock when re-entering the simulation runtime from real code and adding Δ_1 to δ_q to schedule the event with a delay δ'_q. The clock is restarted upon returning to real code and thus δ_2 is accurately computed as $\Delta_1 + \Delta_2$. In addition to safe scheduling of events from simulation code, which can be used to communicate with simulated network and application components, the same technique must be used to allow real code to read the current time and measure elapsed durations.

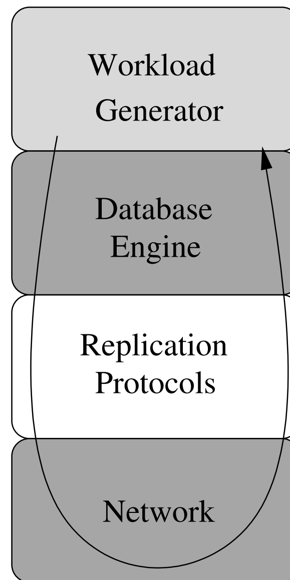


Figure 2.5: Simple overview of the system architecture.

2.2 System Architecture

The system is comprised of four distinct modules: *i*) workload; *ii*) transactional database engine; *iii*) group-based replication protocols; and *iv*) network infrastructure. These are depicted in Figure 2.5 as a stack of components. Since the work presented is about database replication, the system is contains several instances of the given stack, one for each database site.

On top of the stack one finds the workload generator. It mimics clients issuing transaction requests to the database engine. Transactions submitted to the database engine are executed and if any item is updated during its execution, then it cannot commit before the replication protocol disseminates the updates to the other sites. Therefore, the database engine interacts with the replication protocol in order to propagate the updates. This is done accordingly to the replication protocol in use. Once the replication protocol propagates the updates to all other replicas using the network infrastructure, it can then tell the database that it has terminated and the transaction is ready to be committed. Read-only transactions do not need to synchronization among replicas, therefore they can commit without any interaction with the replication protocol.

The system under study are the replication protocols, therefore real implementations are used in the replication part of the stack. Database and network are abstracted as simulation models. The workload generator is actually half real, half abstraction, because at its core lies the same collection of distributions and used to generate transactions either for a real database as for the abstraction presented in the model. In order to use it in conjunction with the simulated database a simple adaptor is needed to translate real transactions requests into simulated transactions. This is clearly depicted in Figure 2.5, which presents in a box with white background the real implementations, in the dark background boxes the simulation abstractions and in the light shaded background box the workload generator. The arrow represents an update transaction execution flow.

Using a realistic traffic profile submitted to the system under test is most valuable, since it enables realistic testing of the database engine, the replication protocols and the network. It also allows the system under test to face real world scenarios in which several minor details always arise when deploying a system that has only been tested with toy applications.

Transaction	Probability	Description	Read-only?
New Order	44%	Adds a new order into the system.	No
Payment	44%	Updates the customer's balance, district and warehouse statistics.	No
Order Status	4%	Returns a given customer's latest order.	Yes
Delivery	4%	Records the delivery of orders.	No
Stock Level	4%	Determines the number of recently sold items that have a stock level below a specified threshold.	Yes

Table 2.1: Transaction types in TPC-C.

The next sections detail the architecture briefly described in this section, in a top-down approach according to the picture presented.

2.3 Simulation Model

2.3.1 Workload Model

The application model is defined by the Transaction Processing Performance Council, TPC-C [46]. It is the industry standard on-line transaction processing (OLTP) benchmark which mimics a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. The database contains the following tables: *(i) warehouse*; *(ii) district*; *(iii) customer*; *(iv) stock*; *(v) orders*; *(vi) order line*; *(vii) history*; *(viii) new order*; and *(ix) item*. The traffic is a mixture of read-only and update intensive transactions. A client can request five different transaction types as follows: New Order, adding a new order to the system (with 44% probability of occurrence); Payment, updating customer's balance, district and warehouse statistics (44%); Order Status, returning a given customer latest order (4%); Delivery, recording the delivery of products (4%); Stock Level, determining the number of recently sold items that have a stock level below a specified threshold (4%). TPC-C table related information and transaction pattern are best depicted in Table 2.1 and Table 2.2. The model accurately follows database scaling rules defined by TPC-C and the database is scaled according to the number of clients. Namely, an additional warehouse should be configured for each additional 10 clients and initial sizes of tables are also dependent on the number of configured clients.

Each client is attached to a database server and produces a stream of transaction requests. When a client issues a request it blocks until the server replies, thus modeling a single threaded client process. After receiving a reply, the client is then paused for some amount of time (think-time) before issuing the next transaction request.

The model abstracts away benchmark requirements such as screen load and background execution, which are not relevant for this work. In fact, these are not significant for the evaluation of replication and group communication protocols, therefore, TPC-C is used as the basis for a realistic application scenario and not as a benchmark.

During a simulation run, clients log the time at which a transaction is submitted, the time at which it terminates, the outcome (either abort or commit) and a transaction identifier. The latency, throughput and abort rate of the server can then be computed for one or multiple users, and for all or just a subclass of the transactions.

Relations	Number of items 100 (Cli.)	2000 (Cli.)	4000 (Cli.)	Tuple size
Warehouse	1×10^1	2×10^2	4×10^2	89 bytes
District	1×10^2	2×10^3	4×10^3	95 bytes
Customer	3×10^5	6×10^6	12×10^6	655 bytes
History	3×10^5	6×10^6	12×10^6	46 bytes
Order	3×10^5	6×10^6	12×10^6	24 bytes
New Order	9×10^4	18×10^5	36×10^5	8 bytes
Order Line	3×10^6	6×10^7	12×10^7	54 bytes
Stock	1×10^6	2×10^7	4×10^7	306 bytes
Item	1×10^5	1×10^5	1×10^5	82 bytes
Total	$\approx 5.1 \times 10^6$	$\approx 10 \times 10^7$	$\approx 2 \times 10^8$	

Table 2.2: Size of tables in TPC-C.

2.3.2 Transaction Processing Model

The database server handles multiple clients and is modeled as a scheduler and a collection of resources, such as storage, cache, processing units and a concurrency control policy [9, 8]. Each transaction is modeled as a sequence of operations, which can be one of: *i*) fetch a data item; *ii*) do some processing; *iii*) write back a data item; *iv*) call replication engine; and *v*) issue a lock related operation. Upon receiving a transaction request each operation is scheduled to execute on the corresponding resource. As an example, the generic replicated transaction execution path, is shown in Figure 2.6. Every resource is modeled as a simple queue. For example, I/O requests arriving at storage will have to wait if the I/O throughput is already at its maximum usage. The same happens in the CPU pool, if a job arrives and no CPU is free, it will have to be put on a waiting queue or preempt the execution of an executing job.

Processing operations are scaled according to the configured CPU speed. Each is then executed in a round-robin fashion by any of the configured CPUs. Additionally, simulated CPUs are also used to schedule real jobs by the centralized simulation runtime. Therefore, transaction execution can be preempted to assign the CPU to real jobs, enabling concurrent execution between real and simulated jobs.

I/O operations are modeled using a storage abstraction that specifies procedures for fetching and storing items. Its service rate (throughput) is defined by the number of allowed concurrent I/O requests and the latency of a single request. Each request manipulates a write attempt, meaning that the cache hit ratio is very close to 100%, hence storage bandwidth becomes configured indirectly.

Operations for fetching and storing items are submitted to the concurrency control module (lock manager). Depending on the policy being used, the execution of a transaction can be blocked between operations. Items get locked accordingly to the user defined locking policy. The concurrency control may be based on timestamp or strict two phase locking (2PL) protocols [14]. In a timestamp locking policy, items fetch for reading are ignored, while items updated are exclusively locked. When a transaction commits, all other transactions waiting on the same locks are aborted due to write-write conflicts. In the strict two phase locking all items get locked. Whenever a transaction aborts, its locks are released and can be acquired by any subsequent transaction. In addition, all locks are atomically acquired, and atomically released when the transaction commits or aborts, thus avoiding the need to simulate deadlock detection. This is possible as all items accessed by the transaction are known beforehand.

The transaction processing model also offers hooks to plug in replication protocols. This is actually

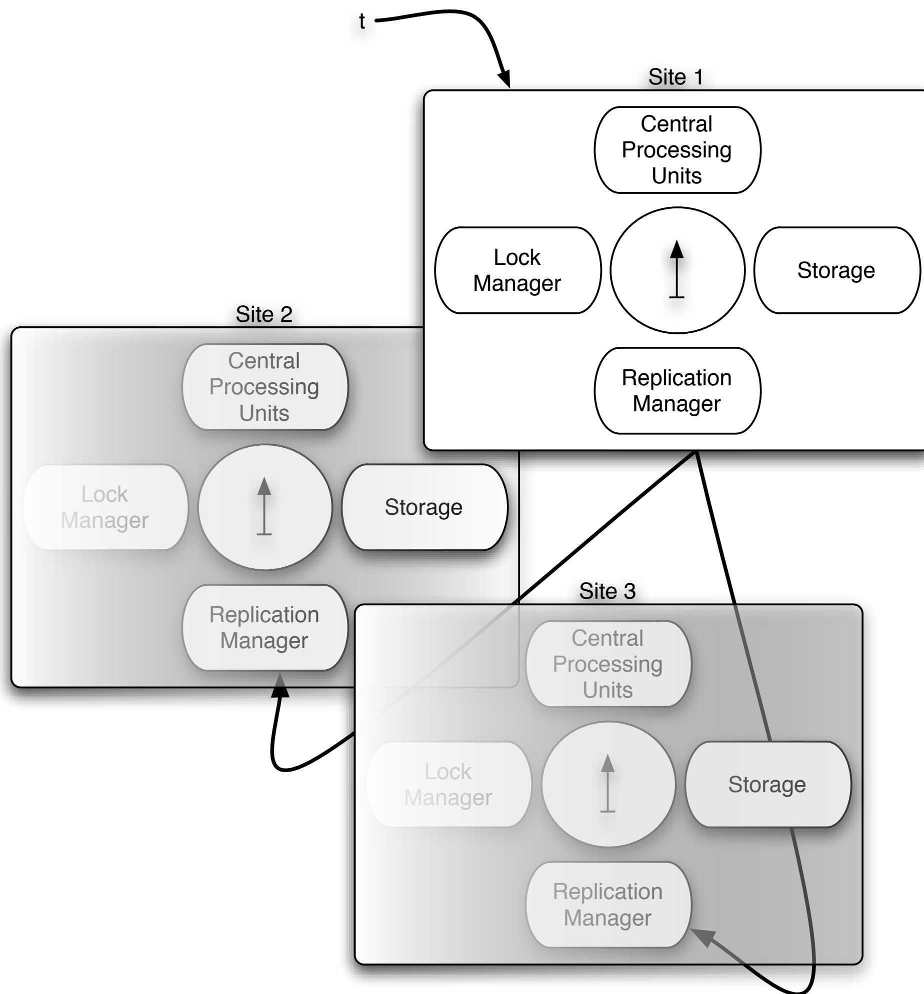


Figure 2.6: Simulated transactions and replicas interaction.

one of the main interfaces between the simulated environment and the system under test (i.e. replication protocols). In detail, these offer the possibility to: *i*) intercept queries upon arrival, to allow classification procedure and ordering required by the conservative protocols; *ii*) intercept read and write-sets upon entering commit and producing realistic amount of write values which are to be transmitted over the network, hence will produce real bandwidth consumption; *iii*) handle installation of remote updates, for which there is no directly attached client entity. Note that the interaction between simulation and real code is also performed at this level by using the centralized simulation kernel (see Section 2.2).

During a simulation run, the usage and length of queues for each resource are collected and used to examine in detail the status of the server. This is most useful to find contention in the system. For example if the queue at one resource becomes extensive then a bottleneck has been found. Then, one may confirm the bottleneck theory by examining the usage data to verify what is the highest throughput rate of the resource.

Finally, the abstract model of the database has been named as *SSFDb*.

2.3.3 System Under Test

The system under test is composed by actual implementations of database replication protocols an group communication. Three different implementations are used as a case study in this work. Namely, Database State Machine (DBSM), Postgres-R (PGR) and Conservative (CONS) replication protocols.

When considering the conservative protocol, transactions are set to classify prior to their execution. This involves a call to the replication layer in order to classify the transaction. While the classification is not concluded, the transaction is put on-hold. Once it finishes, the transaction may begin its execution. By the time the commit operation is issued, the write values are gathered and sent to the other replicas, hence another call to the replication service is issued. On the other hand, if the DBSM protocol is considered, there is only one call to the replication module. It happens when the transaction tries to commit. This is also the case for PGR, which requires an additional communication step when compared to the DBSM. All in all, every update transaction goes through the replication layer at least once. Therefore, there is one or more interactions between real code prototypes and simulation.

Each of the replication protocols rely on an atomic multicast protocol, which is provided by the group communication layer. Such primitive is implemented in two stages. A view synchronous multicast protocol and a total order protocol. The former, view-synchronous multicast, works in two phases. First, messages are disseminated, taking advantage of IP multicast in local area networks and falling back to unicast in wide-area networks. Then, reliability is ensured by a window-based receiver initiated mechanism similar to TCP/IP [40] and a scalable stability detection protocol [27]. Flow control is performed by a combination of a rate-based mechanism during the first phase and the window-based mechanism during the second phase. View synchrony uses a consensus protocol [42] and imposes a negligible overhead during stable operation.

The goal of the stability detection protocol is to determine which messages have already been delivered to all participants and can be discarded from buffers. It is therefore a key element in the performance of reliable multicast. Stability detection works in asynchronous rounds by gossiping: *i*) a vector S of sequence numbers of known stable messages; *ii*) a set W of processes that have voted in the current round; and *iii*) a vector M of sequence numbers of messages already received by processes that have voted in the current round. Each process updates this information by adding its vote to W and ensuring that M includes only messages that have already been received. When W includes all operational processes, S can be updated with M, which now contains sequence numbers of messages discovered to be stable.

Total order is obtained with a fixed sequencer protocol [15, 28]. In detail, one of the sites issues sequence numbers for messages. Other sites buffer and deliver messages according to the sequence numbers. View synchrony ensures that a single sequencer site is easily chosen and replaced when it fails. By implementing total order within the prototype, it becomes possible to later explore several optimizations of atomic multicast in the context of transaction processing. Namely, semantic reliability [38] and optimistic total order [37, 44]. Semantic reliability improves throughput stability in heterogeneous and wide area networks by discarding messages that become obsolete while still in transit, for instance, because a transaction is known to have aborted. Optimistic total order recognizes that it is possible to exploit the spontaneous order of messages which happens with high probability to optimistically start processing transactions. If the order turns out to be wrong, the execution is rolled back and the transaction is redone in the correct order.

2.3.4 Network Model

The network model is defined by using the components from a library of components that can be reused. This is the case of the SSFNet [20] framework, which models network components (e.g. network interface cards and links), operating system components (e.g. protocol stacks), and applications (e.g. traffic generators). Complex network models can be configured using such components, mimicking existing networks or exploring particularly large or interesting topologies. The SSFNet framework provides also extensive facilities to log events. Namely, traffic can be captured in the same format used in real networks and thus the log files can be examined using a variety of existing tools.

The interface between the group communication and the network is the other main interface between the system under test and the simulated environment. It offers a simplified version of the standard socket API for connectionless protocols. Again, interaction between simulation and real code is performed at this level by using the centralized simulation kernel (see Section 2.2).

2.4 Simulation Kernel

In the previous section prototypes of the replication and group communication protocols were said to be embedded within simulation. This section explains how this is achieved by detailing the simulation kernel and how it handles real time execution. It starts by comparing a real system and a simulation model. Then it goes on demonstrating how to mix both using the simulation kernel developed.

2.4.1 Real-Time

Running selected components with simulation models of realistic environments, workloads, and faultloads is accomplished using a centralized simulation kernel [11]. This approach is implemented as a small extension to the Scalable Simulation Framework (SSF) [19] specification that greatly simplifies interfacing real implementations within simulation models while accurately reproducing the timing behavior of real systems.

The extension of the SSF interface is very simple and consists on being able to designate selected entities as *real-time entities* by overriding the isRealTime() method. A process associated with such entity becomes a *real-time process* and behaves as follows: The profiling clock is started when the process reads an event from an incoming channel and stopped whenever the process writes an event to an outgoing channel. Code executed between writing an event and reading another event is therefore not accounted for. After stopping the profiling clock, the real-time process is not rescheduled until simulation time has advanced by as much time as real execution took. The event is actually written to the channel only after simulation time has catch-up. The proposed programming interface was implemented in Java as the *MinhaSSF* package. This was required as source of the existing implementation is not available.

Accounting real-time within the simulation kernel is easily implemented in existing SSF implementation as this boils down to taking into consideration real time processes whenever entering the simulation runtime and minor changes to the scheduler. A key issue is the profiling clock used to measure real-time. It is important that the method used allows a fine-grained measurement of time by one operating system thread even if other concurrent threads are running and can thus preempt the desired thread. This is achieved in the Linux operating system using the perfctr patch [39], which offers a virtualized hardware cycle counter for each operating system process.

The proposed semantics is targeted at client/server interactions between simulated and real compo-

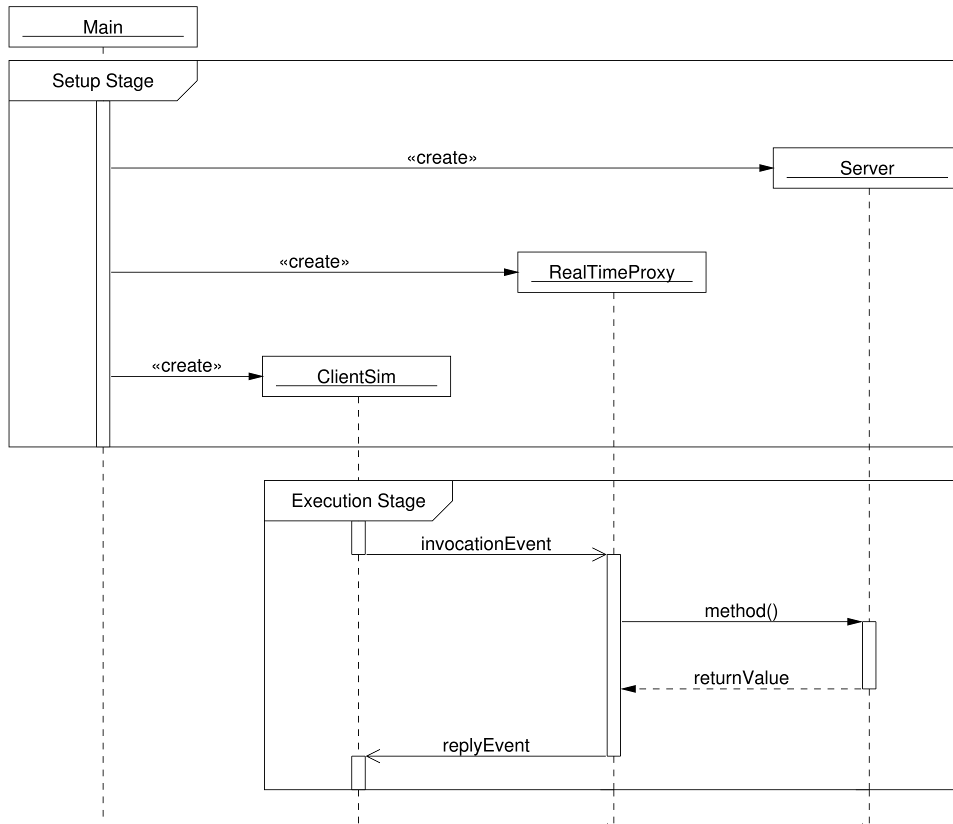


Figure 2.7: Simulated client calls server real implementation.

nents. Real code calling into simulation by means of writing an event will suspend accounting of real time, which is resumed upon reading the return event. Conversely, when behaving as a server, accounting of real time starts upon receiving an invocation event from simulated components and until a return is written.

The SSF specification as well as the extensions mentioned in this section are implemented in Java and available to the GORDA project[18] and are to be release in the near future as open source.

2.4.2 Client/Server Utility Classes

To avoid the tedious and error prone task of manually writing stubs for every interaction, i.e., which translate events into invocations and back, a set of proxy classes has been implemented. These were realized using Java reflection and provide a simple and clear interface that enable generalized usage and complete isolation between simulation models and real implementation. A proxy, named *Simulation-Proxy*, is responsible to transform an invocation made by real code into a pair of events (write and read operations), that transparently interface with simulation code. The second proxy, name *RealTimeProxy*, does the reverse operation, listening for events, invoking real code and replying the result.

The role of dynamic proxies is better understood with an example. Recalling the previous client/server example from Section 2.1.2, one is able to easily reuse the real server implementation in conjunction with the simulated client, as Figure 2.7 illustrates. Keep in mind that entities and channels are implicit in the diagrams, as already previously stated. The server implementation is embedded into a RealTimeProxy, which in its turn is a MinhaSSF real-time entity. This proxy contains a process that reads incoming

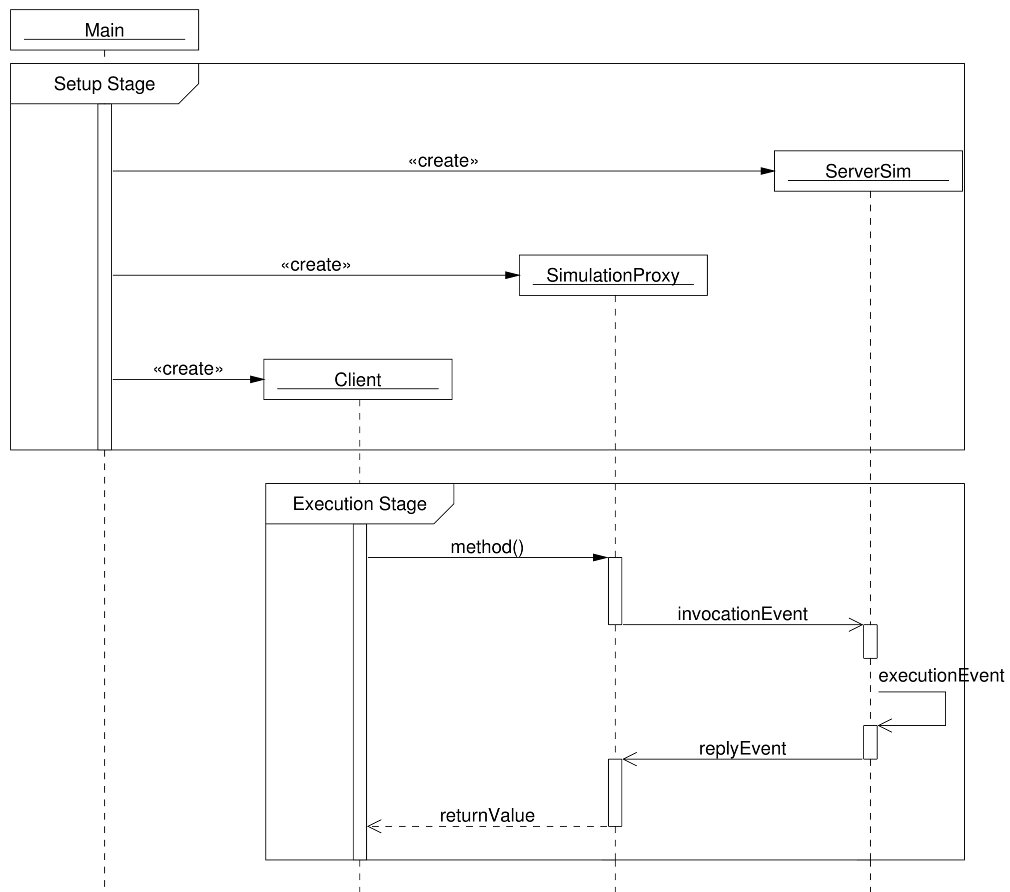


Figure 2.8: Client real implementation calls simulated server).

events from an invocation channel and translates them into calls to the server implementation. At the same instant the process reads the invocation event, the simulation kernel implicitly starts a timer which keeps track of time spent in the real invocation, which is the next thing the proxy performs after reading the event. When the method returns, the proxy writes a reply event containing the result into the reply channel and the kernel implicitly suspends the timer. The reply event, which is delivered to the simulated client, after the time in the real execution elapses in the simulation clock. In the mean time, the RealTimeProxy process is not scheduled again until the simulation clock catches-up.

Calling simulation from real code is also possible due to the other proxy: *SimulationProxy*. Taking the client/server example, one more time, and now considering the client as the real implementation and the server as a simulation model, its usage is depicted in Figure 2.8. The SimulationProxy implements the interface of the server, exposes it to the other objects and provides channel end-points that can be connected to the simulated server to convey invocations and replies. A reference to the proxy object is provided to the real Client instance making it believe it has actually a reference to a server implementation. Once the client calls the *executeRequest*, the proxy translates this call into an invocation event, writes it to the channel, and blocks waiting for the reply. At this moment the MinhaSSF kernel suspends accounting of real time associated with the real client execution. When the server receives the event, which only happens after the simulation clocks elapses the time accounted in the client execution, it simulates the *executionRequest* and sends back the result by writing an event into the reply channel. Upon delivery of the reply event, the proxy collects the result and handles it back to the client as the return value of the fake *executeRequest* method, so control is back in the client implementation. At the same

Resource	Properties
Processor	2 $\times$ 2.4 GHz AMD Opteron
Storage	1 dedicated 100Gb partition
RAM	4 GBytes
Operating System	Linux 2.6.10-1.737 (Fedora Kernel)
Filesystem	ext3
Database Management System	PostgreSQL 8.0

Table 2.3: System specifications.

time the timer is resumed, hence the kernel restarts accounting execution time of client instructions.

2.4.3 Library

The MinhaLib library builds on top of proxies and exposes an API that closely mimics the Java API, thus easing porting of applications as shown in Appendix [**?**]. It is composed by two distinct components: *i*) network and *ii*) concurrency primitives. This components have the purpose to map real calls to simulation events, being the only change needed in the real code application the import statement, that instead import real code network and concurrency Java standard classes, import MinhaLib classes.

minha This package only contains a single class, `UserSystem`, that is a mirror class of `java.lang.System`, with the only purpose to make available current simulation time to applications.

minha.concurrent This package provides the primitives needed to run concurrent applications on top of MinhaLib. This classes implements the `java.util.concurrent` interfaces and also provides the `UserThread` class to allow multithreaded applications in the simulation environment, this is a mirror class of `java.lang.Thread`.

minha.net This package provides the UDP[1] and TCP protocols needed to run networking applications on top of MinhaLib. It also provides the `Application` class that represents a network host.

2.5 Model Calibration

This section presents how the simulation model is calibrated. In detail, how to configure simulation components so they can reproduce accurately a real environment. The purpose of the calibration is to *tune* the model in order to guarantee generalized conclusions.

The calibration procedure is driven by the available hardware for testing, hence the model is configure accordingly. The hardware consists in one HP Proliant dual Opteron processors machine. It has 4 GBytes of RAM memory and uses a fibre-channel attached storage with 1 TByte in a RAID-5 configuration. The operating system used is Linux, kernel 2.6.10-1.737, from Fedora, and the database engine used is PostgreSQL v8.0, having all data stored in a dedicated partition sized in 100Gb, hence no other processes perform I/O into that device. Emulated TPC-C clients are configured to execute from remote machines and communicate with the DBMS over a LAN.

Database calibration is performed using the described hardware and running a benchmark with only one emulated TPC-C client. The logs collected are used to tune the simulation model. It is adjusted in

[1]User Datagram Protocol

two distinct levels: *i*) it needs to be setup according the hardware specifications of the real system (the same number of CPUs, the same storage throughput, ...); and *ii*) resource consumption must follow the same profile as in the real system (i.e., the CPU must be under the an equivalent load, the number of access to storage must be equivalent, ...). Using the calibration setup, TPC-C runs with 10 clients, in both the real and simulated systems, are compared to verify if they match.

2.5.1 CPU

The CPU abstraction is modeled as queue in which the arrival process and the service time distribution follow an exponential distribution. The service time for each job is configured according to the transaction type. Given this, execution times are determined by measuring the time a transaction actually spends in a real CPU in a real execution. Therefore, simulated transactions spend equivalent times to the ones that a real transaction would spend in the real system.

In order to obtain CPU processing times, transactions need to be profiled. To accurately obtain the amount of CPU time consumed by each transaction, an instrumented version of PostgreSQL was developed. In PostgreSQL, a process, named the *backend*, handles a single transaction execution from start to end. This makes it easy to use the Linux perfctr patch [39] to obtain the time spent processing by reading the virtualized per-process cycle counter.

In detail, a CPU timestamp counter is used, which provides accurate measure of elapsed clock cycles. By using a virtualization of the counter for each process, measurements of process virtual time (i.e. the time elapsed when the process is not scheduled to run is not accounted for) are also obtained. To minimize the influence in the results, the elapsed times are transmitted over the network only after the end of each query (and thus out of the measured interval), along with the text of the query itself. The time consumed by the transaction's execution is then computed from the logs. By examining the query itself, each transaction is classified.

The analysis conducted using the logs of the real execution. The initial one 17 minutes of the run (100 transaction samples) and the aborted transactions are discarded, so warm-up effects are minimized, and the percentile 90% of the remaining, is considered. The 90% percentile eliminates deviating samples which are due to OS overhead (swapping and process scheduling) as well as other system daemons interference. The resulting histograms (Appendix A) provide a hint on the empirical distribution that the collected samples follow. The profiling process also allowed to fine tune the PostgreSQL configuration as well as optimize the TPC-C implementation.

Once the logs are analyzed and the histograms plotted one may intuitively find an empirical statistical distribution describing the processing times required for each transaction type (Table 2.4). Therefore, using the *fitdistr* function from *MASS* package of the R [23, 21] project for statistical computation, one is able to fit the empirical distribution to a theoretical statistical distribution. Details of the fitting process are presented in Appendix A.

2.5.2 Storage

The storage calibration is accomplished by considering the number of items that a transaction accesses for writing. Again, like in the CPU time profiling, the PostgreSQL engine had to be modified so the number of items a transaction tries to update gets logged. This is achieved by creating update, delete and insert triggers that keep track of the items accessed during the transaction execution. Upon a commit request, this information is passed to the logger which flushes the information to a file and frees memory kept for holding this information.

Transaction Name	Empirical Distribution	Estimators	
delivery	normal	mean=143698975.769	sd=2332369.0642
neworder	uniform	min=6450113.77352	max=16829661.7371
orderstatus	normal	mean=1658057.33333	sd=830521.190802
stocklevel	uniform	min=1845659.43141	max=2328872.56859
payment	normal	mean=2261806.48101	sd=213283.617067

Table 2.4: Transaction CPU times distribution and estimated parameters (nanosecond precision).

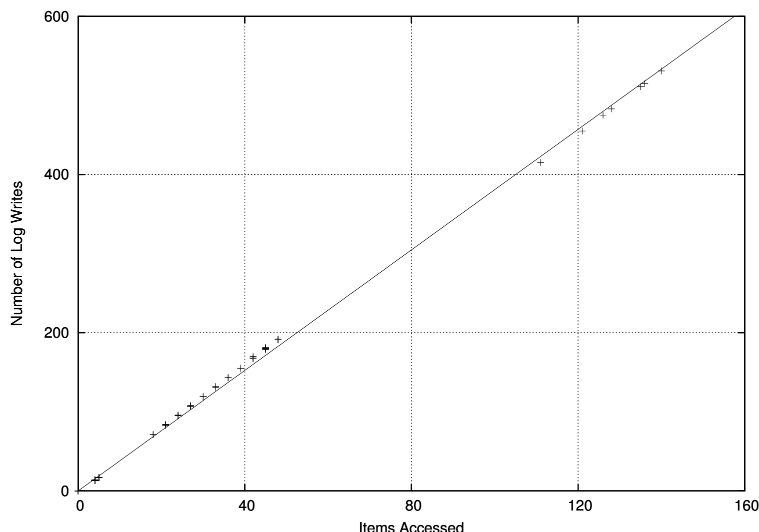


Figure 2.9: *XLogWrite* calls as a function of the number of items accessed.

There are two situations to take into account: *i*) fetching; and *ii*) writing an item. The latency of reading operations is minimal judging from the results obtained, in the tests performed. In fact, despite the warm-up period, the cache hit ratio tends to 1.0, meaning that zero time is spent in I/O for reading. Writing operations requires studying the writes performed in the database log file and the writes performed in the database data space. Writing to the log is critical because it implies at least one disk flush when a transaction issues the commit instruction. This has impact in the total transaction execution latency. It has been verified that, writing to the data storage space does not produce significant impact in the final latency. Explaining this behavior is the fact that PostgreSQL keeps a background writer process that is in charge of flushing the dirty data to the database data space. This is performed asynchronously and most often outside transaction execution context, thence it does not count for the total execution latency.

$$Y = m \cdot X + b \tag{2.1}$$

$$m = \frac{Y_2 - Y_1}{X_2 - X_1} \tag{2.2}$$

In order to calculate the storage service time, the time spent on each log write operation is measured. But knowing the latency of a log write operation is not enough, one needs to find the correlation between the number of log writes and the number of items accessed. This need is imposed because simulation only

Parameter	Value	Short Description
m	3.8088	Slope of the ratio line between items and log writes.
logAccessDelay	986 ms	Average *XLogWrite* call latency.
parallelism	8	Parallel log writes.

Table 2.5: Storage simulation model parameters.

access the number of items a transaction from TPC-C writes and not the number of log writes. With this information the storage model is configured using the writes-items ratio, the latency of each access and a concurrent parameter that states how many concurrent accesses may be performed. In order to obtain the number of log writes performed, one needs to export the number of calls to the PostgreSQL *XLogWrite* function and measure its time. By plotting, for each transaction, the number of items accessed *versus* the number of calls to *XLogWrite* one was able to find a linear relation between both. Figure 2.9, shows the number of log writes as a function of the items accessed. It is clear that there is a linear relation between both. The $2D$ Cartesian coordinate system, states that a line is described by Equation (2.1). Since b is where the line intersects the ordinate axis when X is equal to zero, then its value is obviously zero (Figure 2.9). The line slope is determined by the expression given by Equation (2.2). Therefore, if one considers the two given points $X_2 = (140, 531)$ and $X_1 = (13, 4)$ one is able to determine the line slope without any hassle: $m = \frac{531-13}{140-4} \approx 3.8088$. In addition, Figure 2.9, also depicts, as a solid line, the extrapolated equation: $Y = 3.8088 \times X$.

The number of parallel log writes is determined by the average number of calls between two consecutive flush operations. Flush operations force disk I/O. The measured number of log writes without any flush operation was 8.

Table 2.5, presents the storage simulation model final parameters obtained from the calibration process.

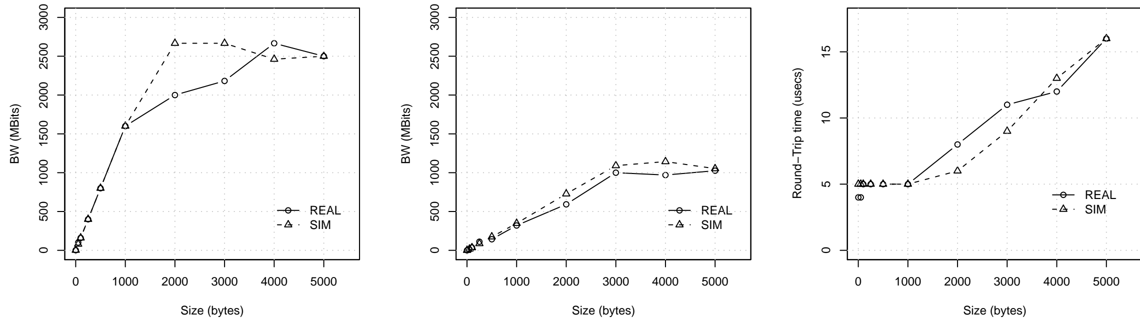
2.6 Validation

Simulation models are a simplification of the real system they mimic. The model presented is no different, therefore the match between the real system and the model presented is by no means an absolute match. Nonetheless, a validation procedure is presented in this section, in order to stress the fact that the developed model performs quite similar to the real system.

Network model and its configuration are validated by comparing the resulting performance measurements of the model to those of the real system running the same benchmark.

Figure 2.10(a) shows the maximum bandwidth that can be written to an UDP socket by a single process in the test system with various message sizes. Figure 2.10(b) shows the result of the same benchmark at the receiver, limited by the network bandwidth. Finally, Figure 2.10(c) shows the result of a round-trip benchmark. The difference observed with packets with size greater than 1000 bytes is due to SSFNet not enforcing the Ethernet MTU in UDP/IP traffic. Deviations from the real system are avoided by restricting the size of packets used to a safe value.

The last stage of the calibration process is to determine how close to the real system the simulation models performs. Taking the configuration values obtained from the storage and CPU calibration, one is able to perform a simple run on the real system and compare the outcome with the same run on the simulation model. The metrics used for comparison are: transaction interarrival and execution latency.



(a) Bandwidth written. (b) Bandwidth read over Ethernet 100. (c) Average round-trip.

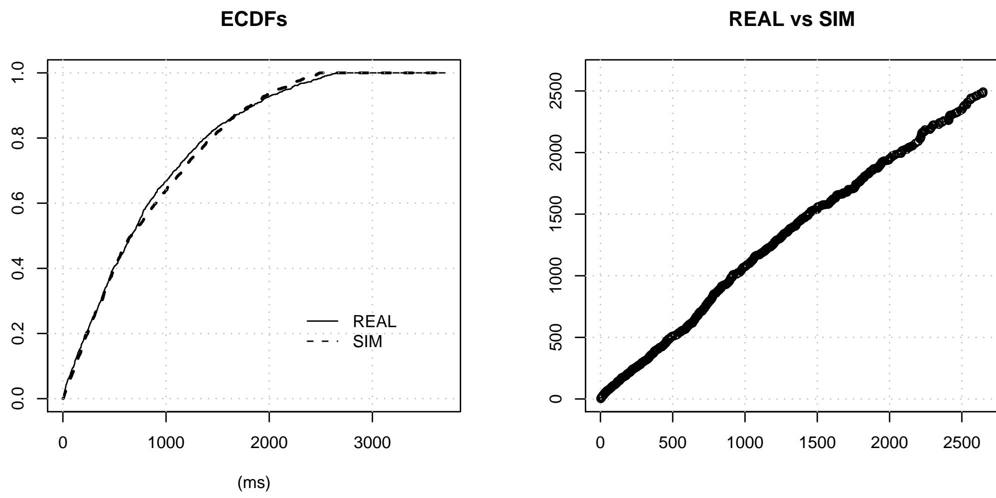
Figure 2.10: Validation of the centralized simulation runtime.

Transaction	**Average TPM**		**Average Latency (ms)**	
	Simulation	Real	Simulation	Real
delivery	2.64	3.66	151.3	192.2
neworder	33.37	32.43	12.82	11.57
payment	29.88	30.14	3.21	3.23
orderstatus	3.15	3.09	1.63	1.21
stocklevel	2.87	3.18	2.08	2.63
TOTAL	71.62	72.93	7.328	6.97

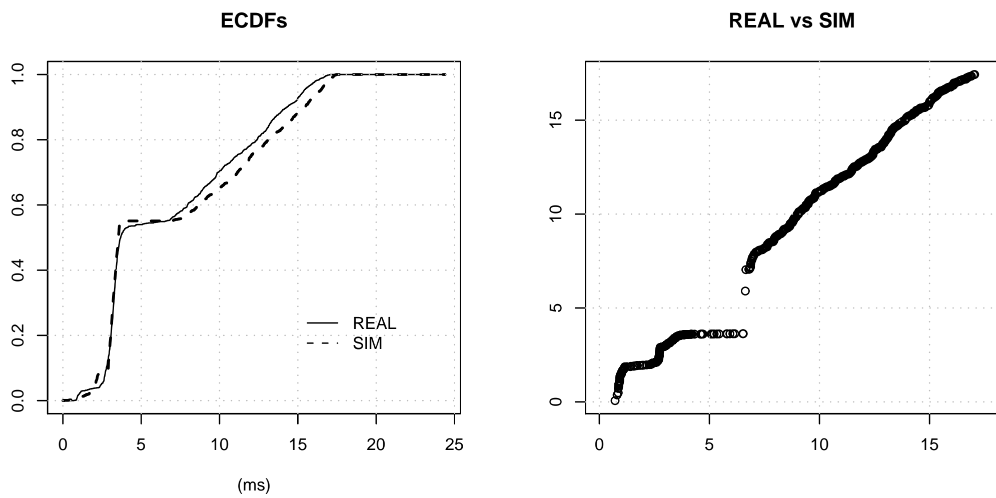
Table 2.6: Simulation vs Real: TPM and latency results.

The former indicates the throughput, while the latter shows how much time a transaction spends inside the database engine during its execution. The abort rate is not taken into consideration, because is its practically zero in both runs.

The validation scenario considers ten emulated TPC-C clients issuing transactions requests to the database. Figure 2.11(b) and Figure 2.11(a), present the comparison of the latency and interarrival of transactions considering the percentile 90%. Note, that these plots show the Empirical CDF of the two runs and present a Q-Q plot to determine how close are each other. In a Q-Q plot, a 45° degrees line states that both plots are perfectly matching. Taking the Q-Q plot into consideration, both are very much alike either in terms of interarrival as in terms of latency, despite the deviation for a small amount of samples in the latter. This deviation is not really important due to the small amount of samples. Table 2.6, depicts numerical values for the average TPM and latency. One may find that real and simulated runs perform real close, hence the system becomes calibrated and validated. For additional details (per transaction analysis), please refer to Appendix B.



(a) Interarrival.



(b) Latency.

Figure 2.11: Simulation vs Real comparison.

Chapter 3

Protocol Evaluation

This chapter we are interested in leveraging the proposed evaluation framework to compare protocols such DBSM [35] and Postgres-R [29], including versions providing snapshot isolation [22, 50], RAIDb [17], NODO [34], and WICE [25].

In fact, we are interested in comparing key features of these protocols. The most relevant is optimistic vs. conservative synchronization. The optimistic categories embraces all protocols that rely on an *apriori* transaction execution without interaction among replicas and a final step of certification immediately before it commits. This is the case for the Database State Machine (DBSM) and Postgres-R (PGR) protocols. WICE protocol is also in this category because it is based in DBSM at its core. The conservative category embraces the RAIDb and the NOn-Disjoint conflict classes and Optmistic multicast (NODO) protocols. These do not allow transaction execution before a coordination is performed between the intervening sites in order to agree on which items they exclusively access on a specified logical time.

A second issue is the granularity of synchronization. This is relevant to RAIDb and NODO, which require conflict classes to be determined *a priori*, but also for an optimistic protocol that ships read-sets to remote certification (i.e. DBSM [35]). The useage of snapshot isolation as a consistency criteria is also applicable to all protocols, regardless of optimistic execution. Finally, we are also interested in the impact of latency in wide area networks, as well as the impact of faults and performance perturbations injected in the group communication layer.

A summary of how each configuration tested represents a combination of these key aspects of target protocols is presented in Table 3.1.

3.1 Motivation

As stated earlier (Section 1.1), there are several constraints that do not allow testing and evaluation to be performed in large scale and in a flexible manner. These problems range from the impossibility of setting up the required scenarios, to the need of having to deal with uncontrollable context variables and lack of deterministic execution. Using the simulation framework described in Section 2, one is able of surpass these problems. Hence, the following sections present an evaluation study of replication protocols showing the benefits of such framework.

The two families of protocols presented in this chapter are named: optimistic and conservative, depending on agreeing on order prior or after execution. We consider also an optimization of the optimistic

Name	Consistency		Granularity		Category		Faults
	Serial.	Snap. Iso.	Coarse	Fine	Optim.	Cons.	
DBSM-G	x		x		x		—
DBSM-g	x			x	x		—
DBSM-g bursty	x			x	x		bursty drops
DBSM-g random	x			x	x		random drops
DBSM-SI		x	x		x		—
DBSM-SI bursty		x	x		x		bursty drops
DBSM-SI random		x	x		x		random drops
PGR		x			x		—
WICE		x		x	x		—
CONS-G	x		x			x	—
CONS-g	x			x		x	—
CONS-SI		x	x			x	—

Table 3.1: Protocol naming and characterization details.

protocol for interconnected clusters in wide-area networks. There are relevant issues worth studying in all protocols presented. For instance:

- Is the replication protocol scalable, i.e., is the system capable of operate under heavy loads, and in a large cluster?
- How does performance degrade when going from a local area network into wide area network, specially regarding the increase on latency for message transmission?
- What is the impact on performance if messages get lost in the network, and which transactions will suffer most in the presence of faults?
- How different does the system perform when comparing Write/Write (Snapshot-Isolation) and Read/Write consistency criteria?

When considering optimistic replication protocols, one is also interested on the following:

- Does network become a bottleneck when sending read sets between replicas and if so, is there a way to reduce its size? What is the impact on the certification process when compressing the read set?
- Which transactions end up aborting most frequently and why?
- Do large transactions end up aborting most due to their longer execution latency?
- Does PGR benefits from not sending the read set over the network at the cost of an additional communication step?

Conservative replication protocols pertinent issues are:

- How should one consider the traffic profile to determine conflict classes?
- What is the impact on performance when defining coarse grained conflict classes (e.g., table level conflict classes)?
- How does contention in conflict class queues increase transaction latency?
- Is transaction latency directly proportional to contention due to class-level conflicts?

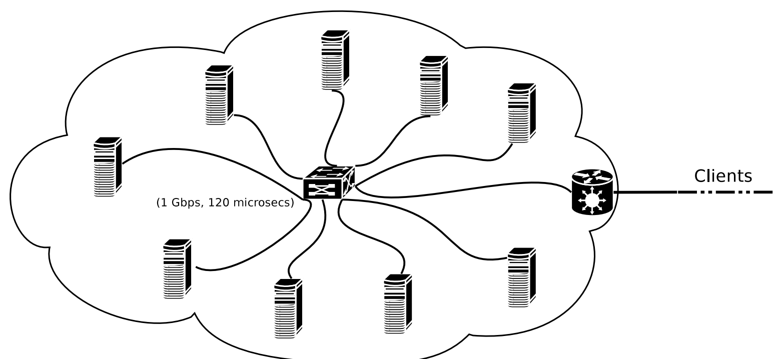


Figure 3.1: Local area network configuration.

3.2 Scenarios

Given the motivations in the previous section, an testing and evaluation work, allowing the answer of such questions, is presented in this chapter. It aims at providing an extensive empirical observation of how the replication protocols under study, behave under different environments. The selected application model is based on one of the TPC set of benchmarks, and has already been exhaustively described in the Section 2.3. In order to replicate the TPC-C database nine sites are considered and spread over the network. Hosting machines are modelled as HP Proliants with two AMD Opteron(tm) Processor 250 processors with 4GB of RAM. For storage, a fiber-channel attached box with 4×36GB SCSI disks in a RAID-5 configuration is considered. Details on how the simulation model calibration is accomplished may be found in Section 2.5.

Each site handles transactions submitted by clients in a multi-master way. The load is symmetric, i.e. every site has the same number of clients connected to it, and every client is modelled as a sequential process. If a client tries to update the database, issuing an update transaction, it must be replicated to the other replicas, hence an update transaction submitted at site S_0 must also update every other site in the system. For the sake of clearness, transactions submitted by a client at a site will be referred to as *local transactions* and transactions submitted at any site by the replication protocols (remote updates) will be referred to as *remote transactions*.

As for the communication infrastructure, two different approaches are considered. The first approach considers nine sites communicating using low-latency and high bandwidth network channels, similar to a cluster environment. All sites are in the same network, hence they all operated in a LAN infrastructure. Links considered have 1 Gbps of bandwidth and the transmission latency is around 120 μs. Figure 3.1, depicts this network configuration.

In the second approach, sites are spread over a wide area network. From the original nine sites six were chosen are grouped on clusters of three, and each of these clusters exhibit the same properties of the LAN network mentioned previously. Intercommunication between groups is accomplished through a wide area network. Messages sent from one cluster to another, go through a border router which communicates with a backbone router. The communication channels between each border router and the backbone router are configured with 100 Mbps of bandwidth and the transmission latency is close to 100 ms. This means that a network packet that is transmitted between two different clusters performs three router hops, considering only router hops, and its round trip delay is given by Equation 3.1.

$$RTT_{WAN} \approx 2 \times (2 \times ClusterLatency + 2 \times WANLatency) \tag{3.1}$$

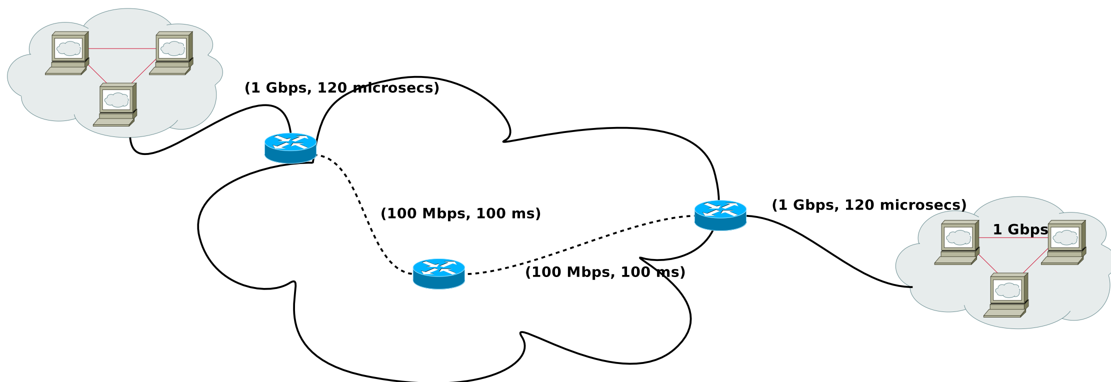


Figure 3.2: Wide area network configuration.

	Serializable			**Snapshot Isolation Level**		
Class./Trans	New Order	Payment	Delivery	New Order	Payment	Delivery
Warehouse	x	x			x	
District	x	x		x	x	
Customer	x	x	x		x	x
Item	x					
Stock	x			x		
Orders	x		x	x		x
OrderLine	x		x	x		x
NewOrder	x		x	x		x
History		x			x	

Table 3.2: Definition of coarse conflict classes for each transaction type in TPC-C.

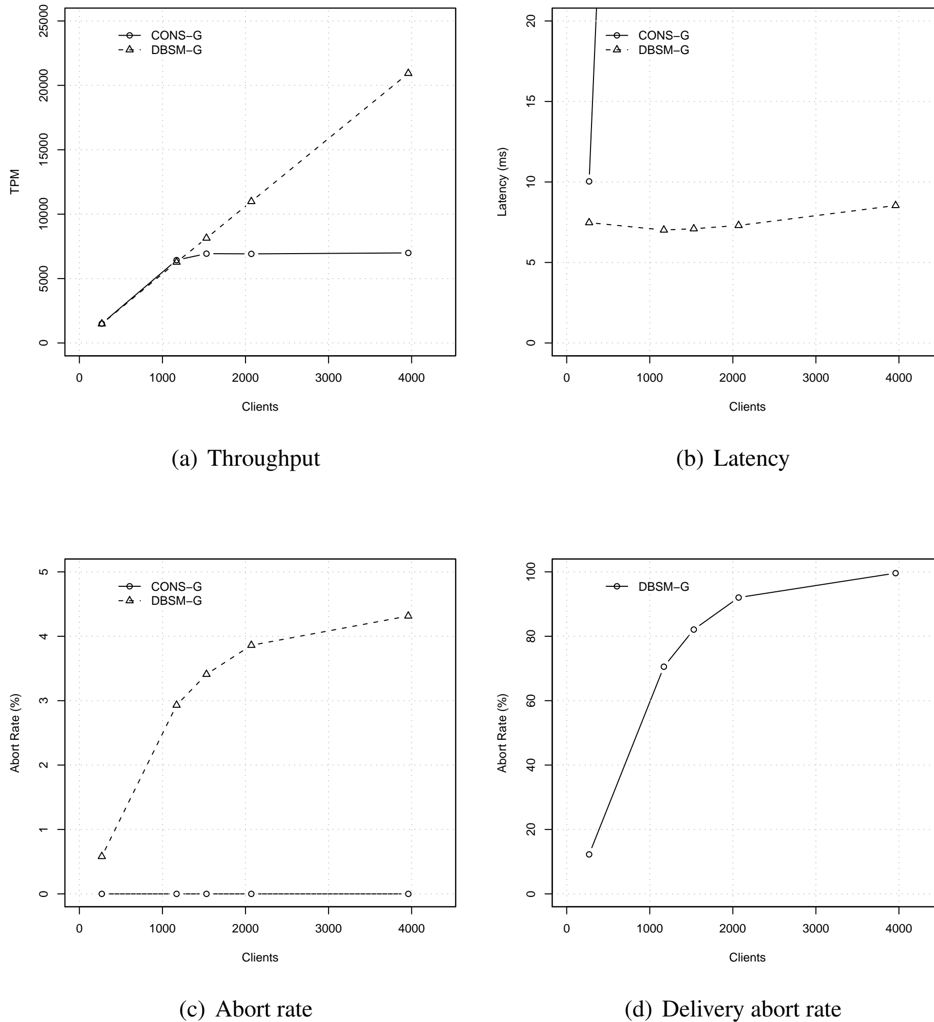
Therefore, accordingly to the configuration described, the transmission time, T, of a network packet between sites on different clusters would be: $T \approx RTT_{WAN}/2 \approx 2 \times (0.12 \times 2 + 100 \times 2)/2 \approx 400ms$. Figure 3.2, depicts this configuration.

3.3 Results: Optimistic vs Conservative

3.3.1 Coarse Grain

The first study evaluates the conservative and the DBSM approaches without exploiting any application specific details and thus in a configuration that can easily be automated. In the conservative approach, each table is considered to specify a conflict class, which can actually be easily extracted from the SQL code. The resulting conflict classes and conflict relations among transactions types are shown in the "Serializable" column of Table 3.2. Regarding the DBSM, special attention needs to be payed to read-set sizes since the propagation of large read-sets may be impractical. An immediate workaround to this problem is to set a limit for the read-set size over which the whole table is used. In the TPC-C, this results in transactions of type Delivery always being marked as reading the entire OrderLine table. All others access only a small number of items.

Figure 3.3, presents performance measurements in the LAN scenario. It can be observed in Figure 3.3(a), that the DBSM protocol with optimistic execution apparently scales much better to a large number of clients than the conservative protocol. As shown by Figure 3.3(b), the bottleneck in the con-



(a) Throughput (b) Latency

(c) Abort rate (d) Delivery abort rate

Figure 3.3: Performance measurements in a LAN with coarse granularity.

servative protocol translates in very large queueing latencies.

This result is highlighted in Figure 3.4 that decomposes latency as seen by a client, and shows that the impossibility to concurrently process transactions that potentially conflict leads to queueing delays which grow very rapidly with the number of clients connected.

However, as seen in Figure 3.3(c), the good throughput of the DBSM is achieved at the expense of a number of aborted transactions. This is especially worrisome since the 4% of transactions being aborted overall are in fact all Delivery transactions as shown in Figure 3.3(d). Therefore, even if such transactions can be resubmitted, there is a very low probability of ever being executed.

Conclusion: Results show that neither DBSM nor CONS protocols scale, without application specific configuration, to a large number of clients with an OLTP load, even with plenty of resources in a LAN.

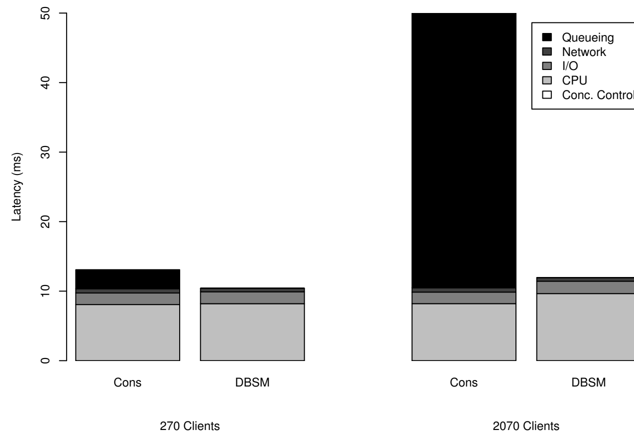


Figure 3.4: Detailed profiling in a LAN with 270 clients.

3.3.2 Fine Grain

To reduce the number of conflicts, a finer granularity has to be specified when defining conflict classes for the conservative approach and the read-set extraction in the DBSM. Fine grained conflict classes are obtained by taking advantage of the fact that all tables except Item have references to the Warehouse table and that clients connected to the same node have high locality regarding a specific subset of warehouses.

Although the simulation run considers this granularity for the CONS protocol, it is impractical since one cannot predict beforehand which subset of tuples, a transaction will access, specially for the NewOrder and Payment transactions. This renders the approach impractical.

In the optimistic protocol, this granularity is practical, because the read-set is extracted during the transaction execution, hence there is no need to predict it. But if there is no hotspots this approach is worthless.

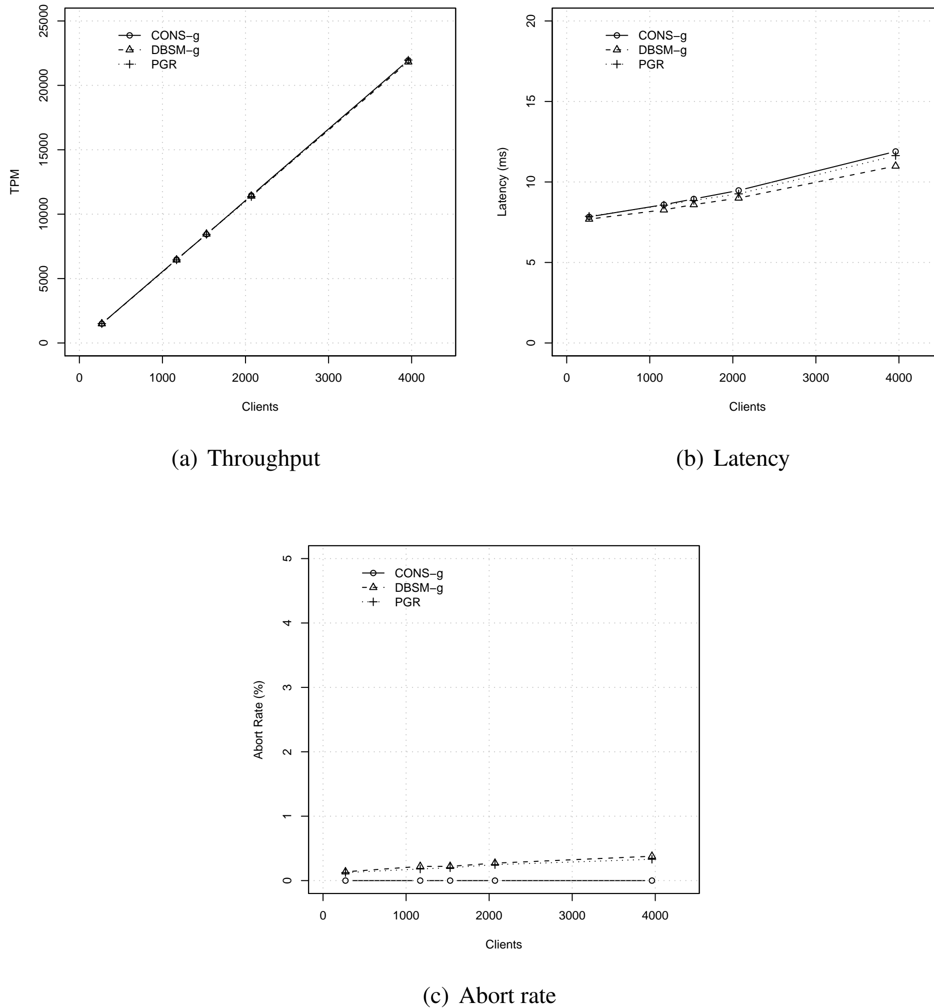
These optimizations are also compared with the PGR protocol which can use the exact read-set by centralizing certification of each transaction. The results are presented in Figure 3.5. It can be observed that all approaches produce approximate results with minimal differences in latency and abort rate. Network usage is also very close, showing that the overhead incurred by the DBSM when sending the read-set is offset by requiring only a single communication step.

Conclusion: When an appropriate can be defined, CONS and DBSM, are equally suited as PGR for an OLTP load in a cluster.

3.3.3 Snapshot Isolation

An alternative approach to avoid synchronization conflicts is to relax the correctness criterion to snapshot isolation [13] which only considers write-write conflicts.

In the DBSM approach, all the concerns previously discussed about the size of the read-set are avoided. As Figure 3.6 shows, it turns out that this alternative has also a benign impact on the performance of the DBSM approach, reducing the number of aborted transactions. Moreover, this is a very appealing alternative, as it avoids all configuration issues. Under snapshot isolation the DBSM and PGR protocols become the same.



(a) Throughput

(b) Latency

(c) Abort rate

Figure 3.5: Performance measurements in a LAN with fine granularity.

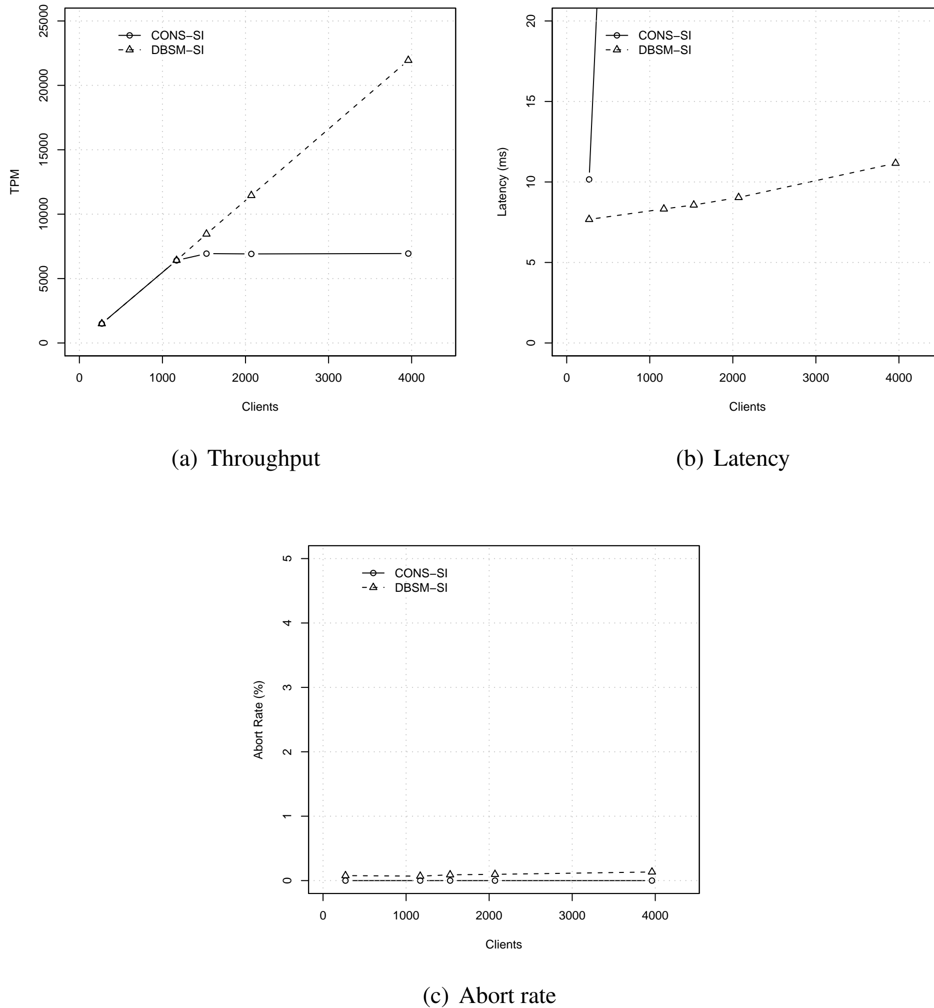
Unlike the DBSM, the conservative approach does not benefit from the snapshot isolation criterion, exhibiting the same latency as in the coarse grain study. In the "Snapshot Isolation Level" column of Table 3.2 the new conflict relations among the transactions are depicted. Regardless of their type, all update transactions still conflict and thus have to be sequentially executed.

Conclusion: When using snapshot-isolation, DBSM presents a reduced abort rate, while the CONS protocol, despite the relaxed correctness criteria, still suffers from conflicts penalties.

3.4 Results: Wide-area Inter-Cluster Enhancements - WICE

It is also interesting to observe how the proposed approaches scale to interconnected clusters in WAN. Wide area networks introduce higher communication latency when compared to LANs. Consequently, transactions take more to conclude, resulting in a higher probability of concurrent transactions in the system, thus higher probability of conflicts.

In a protocol based on distributed locking, the influence of latency can potentially be very large,



(a) Throughput (b) Latency

(c) Abort rate

Figure 3.6: Performance measurements in a LAN with snapshot isolation.

if a node has to wait that all other nodes enter and leave a critical section plus the time it takes to pass the authorization around. In contrast, when using active replication [43, 26], the only overhead is encapsulated in the total order multicast protocol and no additional synchronization is required. Ideally, a database replication protocol based on total order multicast would be able to achieve the same goal.

To study the impact of the communication latency overhead when considering WANs, the DBSM-SI was chosen. Optimizations to this protocol were conducted which resulted in a slightly different protocol named *WICE*.

3.4.1 Performance

The performance of the WICE protocol [25] is evaluated by observing the throughput, latency and abort rate achieved when compared with plain DBSM. As a baseline, we present results obtained by grouping all 6 servers in the same cluster (DBSM CLUSTER). The results are obtained with Write-Write conflict certification, thus achieving 1-SI, are presented in Figure 3.7. Results are presented separately for each cluster.

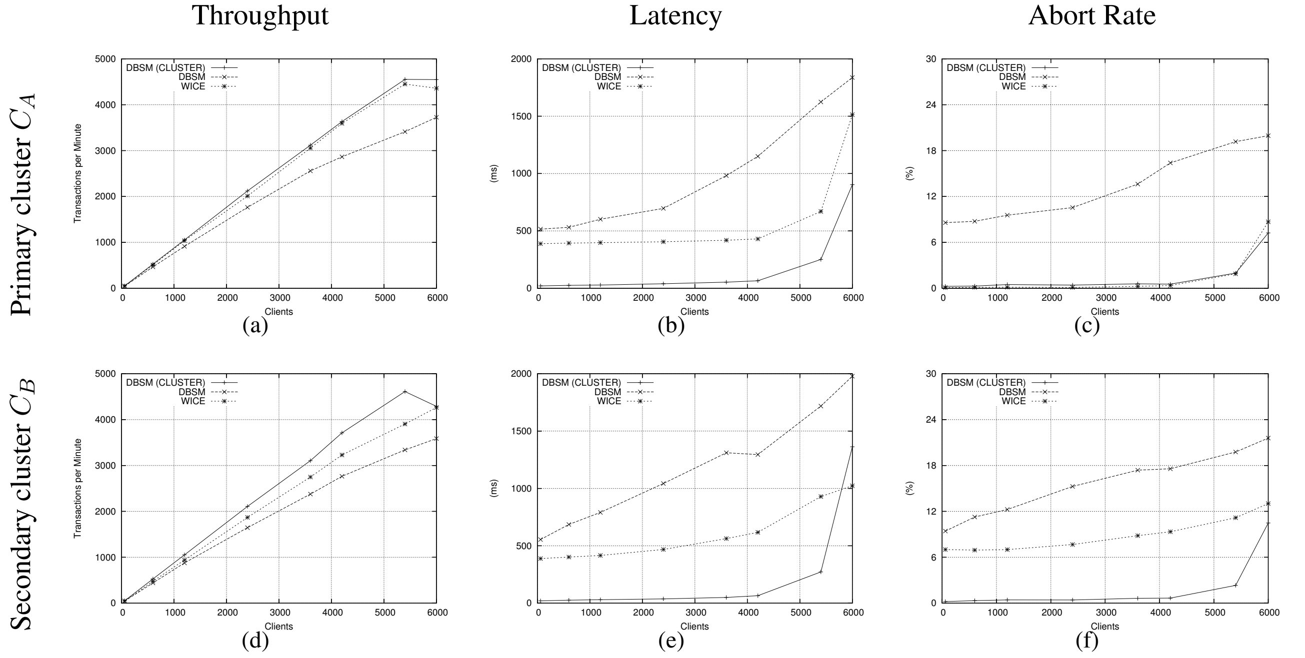


Figure 3.7: Performance results with 1-SI.

The first interesting observation from the baseline protocol (DBSM CLUSTER) is that the capacity of the system is exhausted with 6000 clients. This shows up as throughput peaking (Figure 3.7(a)), increasing latency due to queuing (Figure 3.7(b)), and abort rate due to increased concurrency (Figure 3.7(c)). By examining resource usage logs one concludes that this is due to saturation of available CPU time. We should thus focus on system behavior up to 4000 clients, as a properly configured system will perform flow control to ensure operation in that range. Namely, throughput grows linearly, latency is approximately constant and the abort rate negligible.

Then, we turn our attention to DBSM in the target scenario. Although throughput scalability is apparently close to linear, it is misleading as it corresponds to a high abort rate and a linearly increasing latency, in particular in cluster C_B (Figures 3.7(e) and 3.7(f). Both are explained by the same phenomenon: As locks are witheld during wide area stabilization, queuing delays arise, thus proportionally increasing the probability of later being aborted. Aborted transactions have to be resubmitted by the application, thus further loading the system. It is also important to underline that latency and abort rate impact both clusters equally, as expected, as both suffer with the same $2 \cdot t_{max}$ commit-interval.

As expected, the WICE protocol improves the performance at the primary cluster without negatively impacting secondary clusters. Namely, in the primary cluster the abort rate is negligible (Figure 3.7(c)), comparable only with the DBSM CLUSTER scenario. The latency is also approximately constant in the safe operating range (i.e. up to 4000 clients), although impacted by the round-trip over the wide area link (Figure 3.7(b)). Note however that such impact is very close to the absolute minimum of $2 \cdot t_{max}$ at 400 ms.

Also as expected, the abort rate of transactions initiated in the second cluster, which are impacted by a t_{max} to $2 \cdot t_{max}$ commit-interval, is not negligible although still offering a substantial improvement on DBSM. In the next section, we discuss the impact of this in the expected usage scenario of WICE.

3.4.2 Discussion

The workload assignment used deserves some additional comments. The WICE protocol targets the global enterprise where the goal of replication is twofold. First, by providing a cluster for each region of the globe one avoids having route all queries to a central location and thus avoid imposing the large latency on clients when no updates are performed, while at the same time balancing the load. Second, it improves availability as even catastrophic disasters can only impact the computing or communication infrastructure at a single location. One has therefore to consider clusters located in different timezones, having distinct peak utilization periods.

This means that the evaluation scenario in the previous section, in which traffic in both clusters is exactly the same, is the worst case scenario for the proposed protocol. In reality, one should be able to migrate the centralized sequencer to the currently most loaded cluster. The additional abort rate at other locations can then be easily solved by resubmission, as these clusters are off peak and thus with underutilized resources.

We also have not assumed that resubmission can be done automatically by the database management system. However, this is true for many workloads, especially in current multi-tiered applications. By taking advantage of such option one could thus completely mask the abort rate at secondary clusters.

3.5 Results: Fault-injection and DBSM

In this section the simulation model is used to evaluate the performance and dependability of the DBSM prototype implementation.

3.5.1 Termination protocol performance

The performance of the termination protocol (set of DBSM and group communication instructions) is assessed by conducting simulations in local area with three sites only. Faults are injected and the termination performance is measured.

Fault injection creates adverse conditions causing message losses or high jitter on message processing time. Table 3.3, exhibits a detailed overview of the type of faults injected into the system. Faults are injected by intercepting calls in and out of the runtime as well as by manipulating model state.

The evaluation of the results is two-fold. First, to ensure that all operational sites must commit exactly the same sequence of transactions by comparing logs off-line after the simulation has finished. This condition has been met in face of all types of faults listed in Table 3.3 and ensures the safety of the approach and of the prototype implementation in maintaining consistency. Second, to measure the impact of faults on the performance of the termination protocol and its ultimate consequences at the transaction level. Besides crashes, which as expected have a profound impact in performance by disconnecting a number of clients, the types of faults in Table 3.3 causing more performance degradation are those causing message losses. Figure 3.8, plots the empirical cumulative distribution functions (ECDF) of certification latency with 1000 clients. Note the logarithmic scale in the x-axis. It can be observed that random loss of 5% of messages has much more impact than the same amount of loss in bursts of average length of 5 messages (uniformly distributed). The long tail of the distribution indicates that a small number of transactions is taking as much as 10 times more, in the termination protocol, than before. Table 3.4 shows also an increase in CPU usage by real jobs, showing the extra work by the protocol in retransmitting messages.

Fault type	Parameters	Implementation
Clock drift	rate	Scheduled events are scaled up (i.e. postponed) and elapsed durations measured are scaled down by the specified rate.
Scheduling latency	distribution	A randomly generated delay is added to events scheduled in the future (i.e. in which the process is suspended and scheduled back).
Random loss	rate	Each message is discarded upon reception with the specified probability. Models transmission errors.
Bursty loss	average burst lengths	Alternate periods with randomly generated durations in which messages are received or discarded. Models congestion in the network.
Crash	time	A node is stopped at the specified time, thus completely stopping interaction with other nodes.

Table 3.3: Types of faults injected.

Run	Usage
No Faults	1.22
Random Loss	1.90
Bursty Loss	1.89

Table 3.4: Protocol CPU usage (%).

The impact of faults in the quality of service provided to the application should also be measured by the number of aborted transactions presented in Table 3.5. This is explained by the extra time spent in the termination protocol. In fact, the random loss tail corresponds to the group protocol blocking a few times for short periods during the simulation run. Blocking is caused by a combination of three factors:

- The group protocol enforces fairness by ensuring that each process can only own a share of total available buffering.
- Using a fixed sequencer for ordering messages. This leads to a much larger number of messages being multicast by one of the participant processes.
- Each round of the stability detection mechanism can only garbage collect contiguous sequences of messages received by all participants. As loss is injected independently at each participant, the common prefix of messages received by all processes is dramatically reduced, even with loss rate as low as 5%. This slows down garbage collection.

It is therefore likely that the buffer share of the sequencer process is exhausted and the whole system blocked temporarily waiting for garbage collection. The problem is mitigated by increasing available

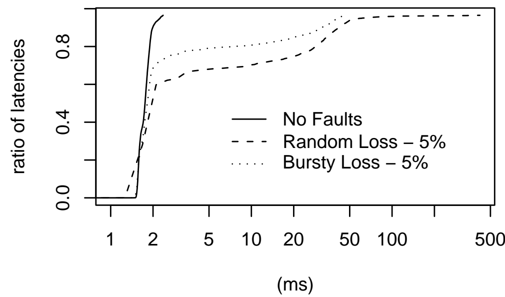


Figure 3.8: Certification latency (fault injection).

	3 Sites/1000 Clients		
Transaction	No Faults	Random Loss - 5%	Bursty Loss - 5%
Delivery	1.41	9.84	4.46
NewOrder	1.46	3.38	1.63
Payment	12.78	22.54	14.15
OrderStatus	2.43	2.93	1.82
Stocklevel	0.00	0.00	0.00
All	6.72	11.94	7.96

Table 3.5: Abort rates with 3 sites and 1000 clients (%).

buffer space or by allocating a dedicated sequencer process. The ideal solution would be to avoiding the centralized sequencer.

Note also in Figure 3.8 that the loss of 5% of messages results in delaying 30% to 40% of messages at the application level. This is a consequence of the total order required by DBSM. This result suggests that relaxing the requirement for total order [36] is necessary for efficient deployment in wide area networks.

Conclusion: The certification performance is not affected in any way due to high jitter on processing messages. On the other hand when there are message losses the certification latency increases 10 times for 20% of the messages. Nonetheless, the degradation of the system as a whole is not directly proportional to the latency increase, as it only shows a slight increase in the abort rate.

3.5.2 Performability: Graceful Degradation

Section 3.5.1 evaluated the termination protocol performance when faults were injected. This section extends the previous one, by assessing how well do the DBSM-g (fine granularity) and DBSM-SI (snapshot isolation) protocols variations degrade under faulty conditions. This is done by conducting a *performability* study.

Performability embraces two concepts: performance and dependability. It is a composite measure which is often used to assess if a system is able to degrade gracefully in the presence of faults. If a system degrades gracefully, it tolerates faults, appearing to be working normally. The study presented in this section relates to the fault injection by dropping messages, either randomly or where there are bursts.

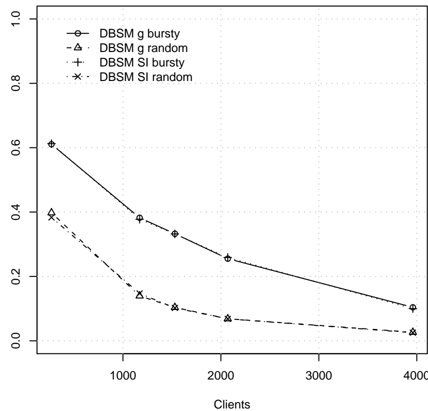


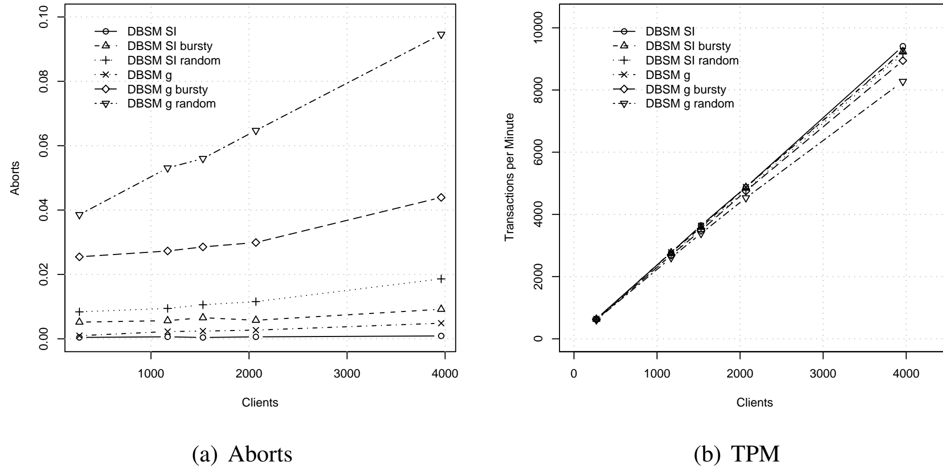
Figure 3.9: P($L \leq \tau$).

The most representative transaction in the TPC-C benchmark is the one that stands for the arrival of a new order to the system. Then, NewOrder transactions are a good indicator on how the system is performing. Therefore, one is interesting in measuring the impact on these transactions when the system is facing adverse conditions. In other words, how graceful does the execution of NewOrder transactions degrade, in the presence of faults, namely when there are packet losses? Having to retransmit the missing packets, implies that the replication latency increases, which directly changes the average response times for the incoming new orders. The performability metric considered to conduct the study is the system's ability to commit a NewOrder within the period of time needed to commit 90%, of the same kind of transactions, when there are no faults at all [45]. Put differently, given L, the average execution latency of a NewOrder transaction, and τ, as the percentile 90 of the commit time in the faultless system, the metric considered is the probability: $P(L \leq \tau)$.

DBSM fine granularity and DBSM with snapshot isolation consistency criteria, both in the LAN environment described in Section 3.2, are the scenarios in which *random* and *bursty* drops of messages are injected. Figure 3.9, depicts the system behavior from the referenced performability metric point of view. As expected, as the load increases, the probability of committing a NewOrder within the τ period diminishes. As the system suffers more from retransmissions, the probability decreases almost exponentially as load is increased. The results show that random drops are worse tolerated than bursty drops. Note that when there is lite load (270 clients) it is almost 20% worse. Under heavy load, the system performability values tend to be the same, almost zero, both in the presence of bursty and random drops.

Although, from the user point of view, an increase in latency, in a milliseconds scale, may not be perceptible, such growth may influence the response of the entire system, directly in terms of abort rate and consequently in terms of throughput. In the case that there are bursty drops, the probability of a NewOrder transaction commits within the τ period decreases from 60% to 10%. In this case, the abort rate for the two different variations of the DBSM protocol remain below the 5%. On the other hand, if one considers random drops, the probability decreases from 40% to almost 5%. Therefore, the abort rate will increase due to the higher probability of the execution exceed given τ. However, note that none of the protocol variations shows an abort rate above 10%, even under heavy load (3960 clients). Figure 3.10(a), depicts the abort rate.

Figure 3.10(b), depicts the throughput of successful new orders per minute that the system is able to achieve. It shows that there is only one situation in which performance degradation is perceptible. This is



(a) Aborts

(b) TPM

Figure 3.10: Performance comparison.

the case of *DBSM-g random*. All the others are not affected considerably in the presence of faults. In fact, in the cases that snapshot isolation is considered the system does not exhibit degradation at all since, the abort rate remains below 2% and the throughput matches the one in which there are no faults. One may conclude that snapshot isolation is more tolerant to latency variations due to network retransmissions than the other variant (fine grained). The reason is that the SI consistency criteria considers only write-write conflicts, therefore, despite the increase of network latency, which leads to a larger period of time in which concurrent transactions may occur, few write-write conflicts do really happen. In the fine grained consistency criteria, read-write conflicts are detected, hence there is a higher probability that a conflict occur, when increasing the network latency.

A final remark to the performability of the random drop scenarios. When looking at the abort rate for the *DBSM-SI* and *DBSM-g*, one may find that there is a difference of almost 7%. This difference should be translated into a smaller value in the probability of committing a NewOrder transaction when comparing both. What is preventing this from happening is that aborted transactions take less to execute, and release their resources sooner. This way, *DBSM-g* system exhibits less contention when compared to the *DBSM-SI*. Less contention means less time that transaction have to wait for resources to become available, hence exhibit smaller execution times. As a consequence, the probability of committing a transaction within the τ period of time becomes the same for the two protocols.

Conclusion: It was shown that for the snapshot isolation variant, the system degrades gracefully, both in the presence of bursty and random drops, despite that in the latter case it suffers most. In the fine grained variant, the performance degradation of the system is not perceptible in the presence of bursty drops. However, when random drops are injected, the system is not able to keep up with the performance of the faultless scenarios.

Chapter 4

Conclusion

This report aims at evaluating database replication protocols by reviving the concept of centralized simulation environment. It builds on the work previously conducted in [11] and proposes extensions to the standard SSF API. These extensions ease the process of setting up and carry on a centralized simulation.

In Chapter 2, the simulation kernel extensions were soundly examined by providing implementation details. Also, a second contribution was introduced. It consists in a Java library of simulation models that abstract a database. Often when evaluating database replication protocols, at a middleware level, the researcher is not interested in the details of the database engine. In fact, he is looking for the engine outputs. Thence, the DBMS may be abstracted as a simulation model and the complexity of the real system is then put aside, because it is not significant for the ultimate purpose. The Java library created provides abstractions which accurately reproduce the behavior of a real DBMS system. It is comprised of several models which mimic central processing units, storage, database engine scheduler, lock manager and replication manager. Model tuning was accomplished by validating and calibrating each subset of abstractions in such a way that results presented show that the simulation is actually comparable to the real system. The calibration procedure was also thoroughly explained in Section 2.5.

When studying database replication, one needs to take into account the network which allows the communication between processes in different sites. This brings forth the need of a network as a mean of message diffusion. The network considered was not physical network. Instead a set of simulation models provided by the SSFNet library were adopted. As opposed to the database models, these did not require validation because it had already been extensively done by the authors.

When moving on to Chapter 3 the reader is presented with an evaluation work that illustrates the performance and reliability of the database replication protocols under study. The evaluation compares key aspects of several replication protocols in different communication scenarios. It comes up with the downside of each kind of protocol and tries to overcome them with some optimizations. Results show that in local area networks, if the protocols are used without being leveraged with the optimizations, the optimistic family of protocols would always ending up aborting a specific kind of transaction (the long ones). On the other hand, the conservative family of protocols would generate very high conflict rates, meaning that queuing effects become dramatic. Therefore transactions would execute in a sequential manner, rendering the system useless due to the latency overhead. With optimizations, all protocols behaved more or less the same.

A performability study was also conducted which assessed the performance degradation, when faults were injected, of two DBSM protocol variations: fine granularity and snapshot isolation. Results show that DBSM fine granularity suffers most when facing packet losses because latency increases, due to

packet retransmission, allowing more concurrent transactions to execute. Such phenomena leads to a higher probability of Read-Write conflicts and consequently to higher abort rates.

As a consequence, the best choice in clusters is likely to be a conservatively synchronized protocols, such as RAIDb or NODO, as these are likely to provide more stable and predictable performance in face of long lived transactions and faults, or DBSM, when the application makes it impossible to determine suitable conflict classes *a priori*. On the other hand, in wide area networks, optimistic execution, especially when coupled with snapshot isolation as in DBSM-SI or with latency masking as in WICE is the best option.

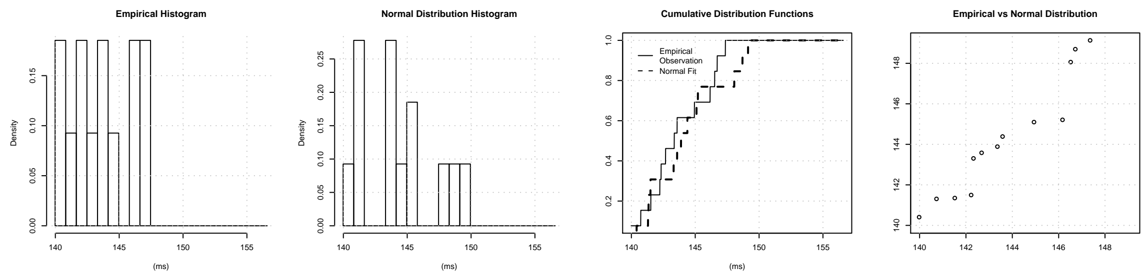
Appendix A

Distribution Parameter Estimation

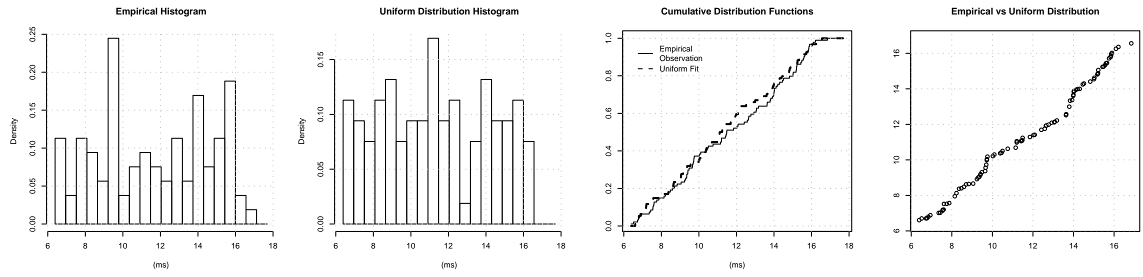
Choosing a generic distribution is accomplished by intuitively fit a the generic distribution to the empirical observation. By looking at the histogram, drawn from the samples, one is able to pick one distribution that the population following.

Having chosen the distribution, the parameters need to be estimated, hence, the *fitdistr* function of the *R software project for statistical computation* is used. This function analyzes the samples and estimates the parameters of the given distribution. In order to assess the goodness of the fit, a Q-Q plot must be drawn. In the Q-Q plots the presented in this section, values of the empirical observation are plotted against values obtain by sampling the fitted distribution. If the plot shows a steady line, 45 degrees steep, than data obtained from the empirical observation comes from a population that follows the given distribution.

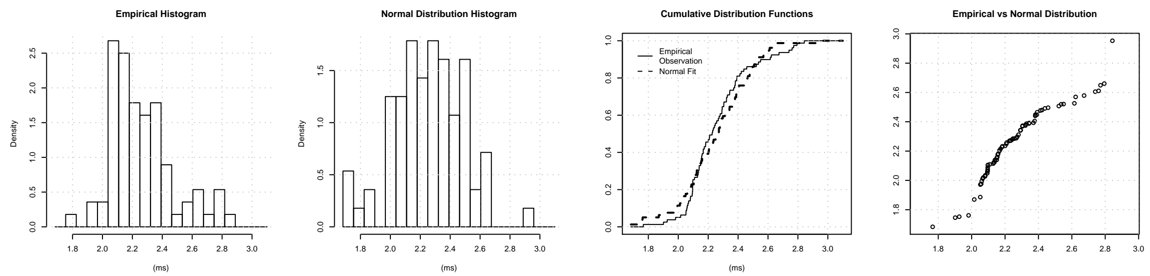
The following figures show the fitting of the CPU and storage times, obtained for each one of the transactions in the TPC-C benchmark.



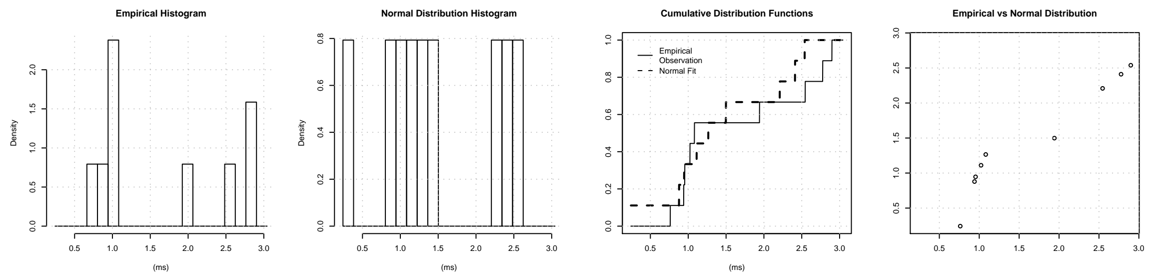
(a) Delivery.



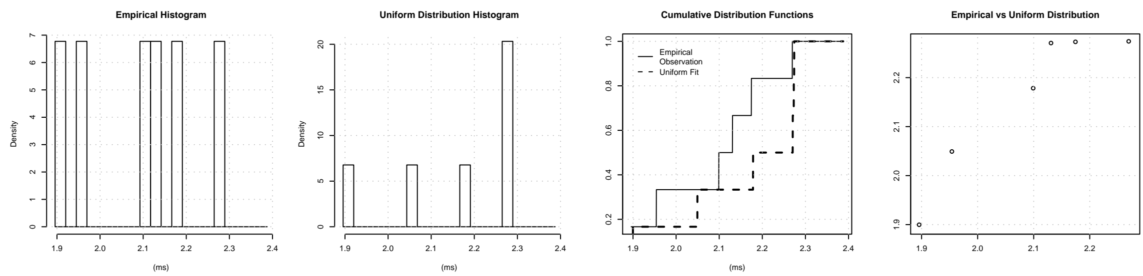
(b) Neworder.



(c) Payment.



(d) Orderstatus.



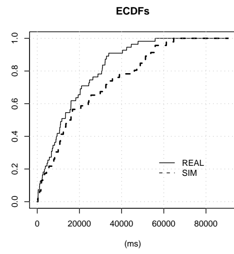
(e) Stocklevel.

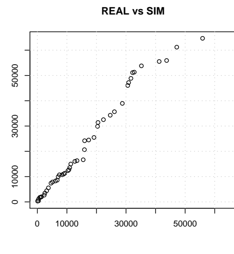
Figure A.1: Processing time fitting.

Appendix B

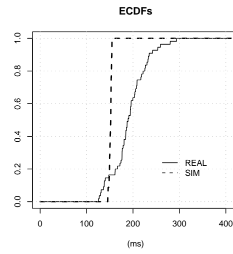
Real vs Simulation Model

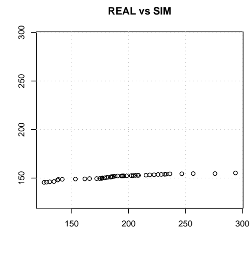
This section details the comparison of the real and simulated run. It compares each transaction in terms of the considered performance metrics: interarrival, latency. These are depicted in the following figures.



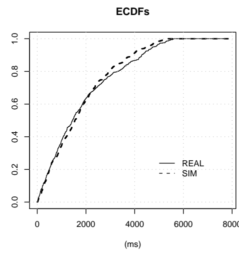


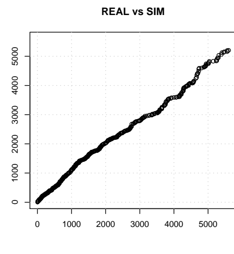
(a) Delivery Interarrival.



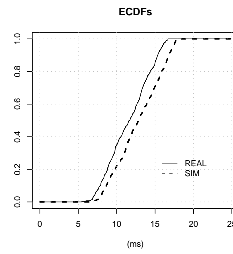


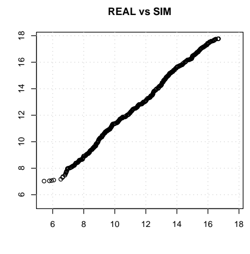
(b) Delivery Latency.



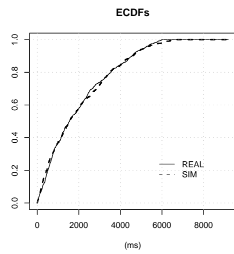


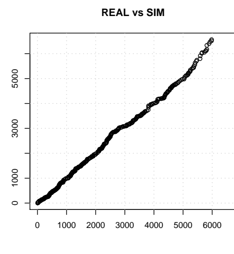
(c) Neworder Interarrival.



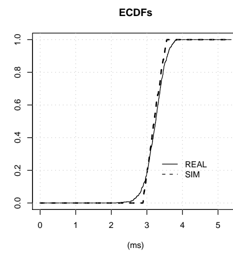


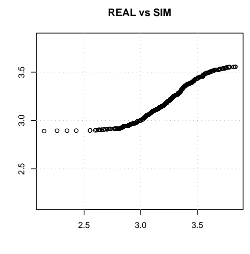
(d) Neworder Latency.



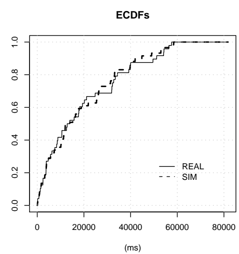


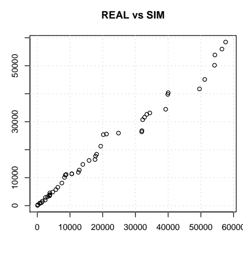
(e) Payment Interarrival.



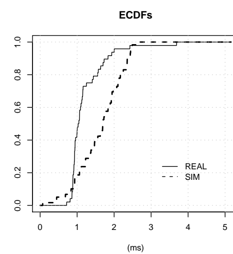


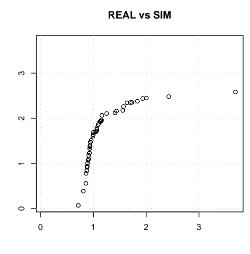
(f) Payment Latency.



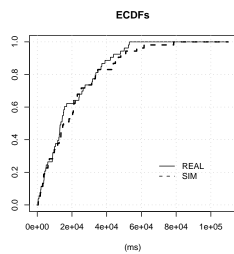


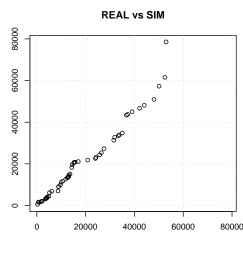
(g) Orderstatus Interarrival.



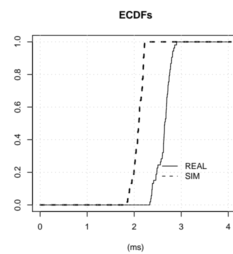


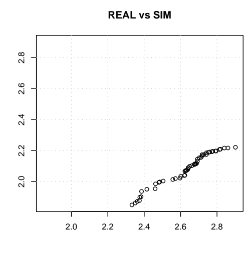
(h) Orderstatus Latency.





(i) Stocklevel Interarrival.





(j) Stocklevel Latency.

Figure B.1: Transaction performance comparison.

Appendix C

Usage Examples

C.1 Start

To start a simulation we run:

```
1  $ java -cp
2  .:lib/cernlite.jar:lib/minhalib.jar:lib/minhassf.jar:lib/ssfnet.jar \
3  SSF.Net.Net 1000000000 dml/AppExample.dml
```

The SSF.Net.Net class will use the services of the DML library to load the content of the AppExample dml file. The `1000000000` indicates the maximum simulation time in seconds.

C.2 DML

In the DML file we have two types of definitions: global and host based. Globally we can define the time resolution (in MinhaSSF is nanosecond), the links between hosts and its delays.

```
1  Net [
2    frequency 1000000000    # 1 nanosecond time resolution (bit on link)

3    link [attach 1(0) attach 2(0) delay 0.005]
```

For each host we can define it's interface bitrate and latency, its application and network stack.

```
4    host [
5      id 1
6      interface [id 0 bitrate 100000000 latency 0.0]
7      graph [
8        ProtocolSession [name app use minha.net.Host
9                    run [
10                         name test.AppExample
11                         arg 0.0.0.2
12                   ]
13           ]
```

```
      ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
      ProtocolSession [name udp use SSF.OS.UDP.udpSessionMaster _find
.dictionary.udpinit]
      ProtocolSession [name ip use SSF.OS.IP]
    ]
  ]
```

The `run` attributes are main class and its command line arguments, like in the real code.

```
$ java test.AppExample 0.0.0.2
```

C.3 Simple Network Hello Application

To show how simple is run a program in MinhaLib, will show the original code and the one using MinhaLib. Two instances of AppExample are created and each one send a `Hello` message to the other.

Real code:

```
package test.real;

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class AppExample {
        public static void main(String[] args) {
                try {
                        int port = 12345;
                        if (args.length < 1) {
                                System.out.println("who is my peer?");
                                return;
                        }

                        DatagramSocket sock = new DatagramSocket(port);

                        // Send something
                        byte[] bytes = "Hello".getBytes();
                        sock.send(new DatagramPacket(bytes, bytes.length,
InetAddress
                                        .getByName(args[0]), port));

                        // Receive something back
                        DatagramPacket recv = new DatagramPacket(new
byte[10], 10);
                        sock.receive(recv);

                        System.out.println("Received " + recv.getLength() +
" bytes at "
                                        + System.currentTimeMillis() + "ms
from "
                                        + recv.getAddress() + ":" +
recv.getPort() + ": "
                                        + new String(recv.getData()));
                } catch (Exception e) {
                        e.printStackTrace();
```

```
32                  }
33          }
34  }
```

MinhaLib code:

```
1   package test.minha;

2   import java.net.DatagramPacket;
3   import java.net.InetAddress;

4   import minha.UserSystem;
5   import minha.net.DatagramSocket;
6   import minha.net.Application;

7   public class AppExample extends Application {
8           public void main(String[] args) {
9                   try {
10                          int port = 12345;
11                          if (args.length < 1) {
12                                  System.out.println("who is my peer?");
13                                  return;
14                          }

15                          DatagramSocket sock = new DatagramSocket(port);

16                          // Send something
17                          byte[] bytes = "Hello".getBytes();
18                          sock.send(new DatagramPacket(bytes, bytes.length,
19  InetAddress
20                                          .getByName(args[0]), port));

21                          // Receive something back
22                          DatagramPacket recv = new DatagramPacket(new
23  byte[10], 10);
24                          sock.receive(recv);

25                          System.out.println("Received " + recv.getLength() +
26  " bytes at "
27                                          + UserSystem.currentTimeMillis() +
28  "ms from "
29                                          + recv.getAddress() + ":" +
30  recv.getPort() + ": "
31                                          + new String(recv.getData()));
32                  } catch (Exception e) {
33                          e.printStackTrace();
34                  }
35          }
36  }
```

As we can see, the import statements related with network were replaced by MinhaLib classes, now the class extends Application and the main method being no more static and current time is given by UserSystem.

Output of `AppExample.dml` run, here we can see all network configuration and initialization steps. We can see that the simulation ends (jump to maximum simulation time) when there no more events to execute.

```
CIDR           IP Block            b16          NHI
--             0.0.0.0/29          0x00000000
0              0.0.0.0/30          0x00000000   1(0) 2(0)

NHI Addr            CIDR Level          IP Address Block    % util
--                  --                  0.0.0.0/29          75.0

**                  Using specified 1.0ns clock resolution


--- Phase I: construct table of routers and hosts


--- Phase II: connect Point-To-Point links


--- Phase III: add static routes
## Net config: 2 routers and hosts
## Elapsed time: 0.092 seconds
** Running for 1000000000000000000 clock ticks (== 1.0E9 seconds sim time)
Booting host /0.0.0.1...
 - starting test.AppExample... Ok
Booting host /0.0.0.2...
 - starting test.AppExample... Ok
Calibrating timer:
 MHz: 2020.170
 cycles/us: 2020
Using perfctr for timing.
Using perfctr for timing.
Received 5 bytes at 149ms from /0.0.0.2:12345: Hello
Received 5 bytes at 153ms from /0.0.0.1:12345: Hello
---------------------------------------------------------------------
[1000000000000000000] MinhaSSF: Simulation time ran out!
---------------------------------------------------------------------
```

C.4 Simple Timer Application

The following code is to show that Java concurrent executors also exist in MinhaLib, in this example like cron tasks.

```
package test.minha;

import static java.util.concurrent.TimeUnit.MILLISECONDS;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;

import minha.UserSystem;
import minha.concurrent.Executors;
import minha.net.Application;

public class AppSchedulerExecutor extends Application {
    private ScheduledExecutorService scheduled_executor;
    private ScheduledFuture task;
    private int count = 1;
```

```
12          public void main(String[] args) {
13          this.scheduled_executor =
14  Executors.newSingleThreadScheduledExecutor();
15          // schedule task
16          task = scheduled_executor.scheduleAtFixedRate(new Task(), 10, 10,
17  MILLISECONDS);
18          // schedule end
19          this.scheduled_executor.schedule(new End(), 100, MILLISECONDS);
20          }

21      class Task implements Runnable {
22          public void run() {
23                  System.out.println("Task (" + (count++) + "): " +
24                                  UserSystem.currentTimeMillis());
25          }
26      }

27      class End implements Runnable {
28          public void run() {
29                  System.out.println("End: " +
30  UserSystem.currentTimeMillis());
31                  task.cancel(true);
32          }
33      }
34  }
```

Output of AppSchedulerExecutor.dml run, with only application output.

```
1   Task (1): 86
2   Task (2): 93
3   Task (3): 103
4   Task (4): 114
5   Task (5): 123
6   Task (6): 133
7   Task (7): 143
8   Task (8): 153
9   Task (9): 163
10  Task (10): 173
11  End: 177
```

C.5 Locks Example

The following code is an usual locks example, without network and dml file.

```
1   package test.minha;

2   import java.util.concurrent.locks.Lock;

3   import minha.UserSystem;
4   import minha.concurrent.ReentrantLock;
5   import minha.concurrent.UserThread;
6   import SSF.Entity;
7   import SSF.process;
```

```
8   class LockUser implements Runnable {
9           Lock l =null;
10          private int idx;
11          LockUser(Lock l, int idx) { this.l = l; this.idx=idx; }

12          public void run() {
13                  while(true) {
14                          UserThread.sleep(1);
15                          System.out.println("[REAL] Locking in "+idx+" at "
16                                          + UserSystem.currentTimeMillis());
17                          l.lock();
18                          System.out.println("[REAL] Working in "+idx+" at "
19                                          + UserSystem.currentTimeMillis());
20                          // just do something to let the clock advance
21                          for(int i=0; i<10000000; i++);
22                          System.out.println("[REAL] Unlocking in "+idx+" at
23  "
24                                          + UserSystem.currentTimeMillis());
25                          l.unlock();
26                  }
27          }
28  }

29  class LocksEmbedder extends Entity {
30          private Lock lock;

31          public boolean isRealTime() { return true; }

32          private class runner extends process {
33                  private int idx;
34                  private runner(int idx) {
35                          super(LocksEmbedder.this);
36                          this.idx=idx;
37                  }

38                  public void action() {
39                          // SETUP

40                          LockUser re = new LockUser(lock, idx);

41                          // EXECUTE
42                          re.run();
43                  }
44          };

45          public LocksEmbedder (int instances) {
46                  this.lock = new ReentrantLock();
47                  for(int j=0;j<instances;j++)
48                          new runner(j);
49          }

50  }

51  class LocksExample {
52          public static void main(String[] args) {
53                  try {
54                          int max=2;
55                          if (args.length!=0)
56                                  max=Integer.parseInt(args[0]);
57                          LocksEmbedder ree = new LocksEmbedder(max);
```

```
                        ree.startAll(1000000000L);
                        ree.joinAll();
                }
                catch(Exception e) { e.printStackTrace(); }
        }
}
```

Output of `LocksExample.java` run.

```
Calibrating timer:
 MHz: 2020.170
 cycles/us: 2020
Using perfctr for timing.
Using perfctr for timing.
[REAL] Locking in 0 at 6
[REAL] Locking in 1 at 7
[REAL] Working in 0 at 7
[REAL] Unlocking in 0 at 43
[REAL] Working in 1 at 44
[REAL] Unlocking in 1 at 70
[REAL] Locking in 0 at 71
[REAL] Locking in 1 at 71
[REAL] Working in 0 at 71
...
[REAL] Working in 0 at 913
[REAL] Unlocking in 0 at 939
[REAL] Working in 1 at 941
[REAL] Unlocking in 1 at 967
[REAL] Locking in 0 at 968
[REAL] Locking in 1 at 969
[REAL] Working in 0 at 969
[REAL] Unlocking in 0 at 998
[REAL] Working in 1 at 998
[REAL] Unlocking in 1 at 1025
----------------------------------------------------------------------
[1000000000] MinhaSSF: Simulation time ran out!
----------------------------------------------------------------------
```

Bibliography

[1] DML specification.
http://ssfnet.org/SSFdocs/dmlReference.html.

[2] Emulab. http://www.emulab.net/.

[3] iSSF homepage. 2003.
http://www.crhc.uiuc.edu/~jasonliu/projects/issf/.

[4] The network simulator - ns-2, 2006.

[5] Planetlab. http://www.planet-lab.org.

[6] SSF research network.
http://www.ssfnet.org/homePage.html.

[7] ACME. Automated configuration management environment, 2006.

[8] R. Agrawal, M. Carey, and M. Livny. Models for Studying Concurrency Control Performance: Alternatives and Implications. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, 1985.

[9] R. Agrawal, M. Carey, and M. Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 1987.

[10] W. Almesberger. umlsim - a uml-based simulator. In *Linux Conference Australia*, 2003.

[11] G. Alvarez and F. Cristian. Applying simulation to the design and performance evaluation of fault-tolerant systems. In *16th Symposium on Reliable Distributed Systems (SRDS'97)*, 1997.

[12] J. Banks, J.S. Carson II, B. L. Nelson, and M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 2000.

[13] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels, 1995.

[14] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[15] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[16] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including os and network aspects. In *In Proc. High-Assurance System Engineering Symp. (HASE'01)*, 2001.

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

[18] GORDA Consortium. Gorda - open replication of databases. http://gorda.di.uminho.pt/consortium, October 2004.

[19] J. Cowie. *Scalable Simulation Framework API Reference Manual*, 1999.

[20] J. Cowie, H. Liu, J. Liu, D. Nicol, and Andy Ogielski. Towards realistic million-node internet simulation. In *Intl. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, 1999.

[21] P. Dalgaard. *Introductory Statistics with R*. Statistics and Computing. Springer, 2002.

[22] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database Replication Using Generalized Snapshot Isolation. 2005.

[23] D. Bates et al. The r project for statistical computing. http://www.r-project.org/, 1997.

[24] Facilita. Facilita forecast, 2006.

[25] J. Grov, L. Soares, A. Correia Jr., J. Pereira, R. Oliveira, and F. Pedone. A pragmatic protocol for database replication in interconnected clusters. In *IEEE Intl. Symp. Pacific Rim Dependable Computing (PRDC'06)*.

[26] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, (4), 1997.

[27] K. Guo. *Scalable Message Stability Detection Protocols*. PhD thesis, 1998.

[28] M. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proc. the* 11th *Int'l Conf. on Distributed Computing Systems ICDCS*, pages 222–230, Washington, D.C., USA, May 1991. IEEE CS Press.

[29] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, 2000.

[30] J. Liu, D. Nicol, B. J. Premore, and A. L. Poplawski. Performance prediction of a parallel simulator. In *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*, 1999.

[31] SPEC logo Standard Performance Evaluation Corporation. Standard performance evaluation corporation (web 2005). http://www.spec.org/web2005/, 2005.

[32] D. M. Nicol and J. Liu. Dartmouth ssf, 2002.

[33] D. M. Nicol and X. Liu. The dark side of risk (what your mother never told you about time warp). In *PADS '97: Proceedings of the eleventh workshop on Parallel and distributed simulation*, 1997.

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

[35] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. *Distributed and Parallel Databases*, 2003.

[36] F. Pedone and A. Schiper. Generic broadcast. 1999.

[37] F. Pedone and A. Schiper. Optimistic atomic broadcast: A pragmatic viewpoint. 2003.

[38] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable multicast: Definition implementation and performance evaluation. *Special Issue of IEEE Transactions on Computers on Reliable Distributed Systems*, 2003.

[39] M. Pettersson. Linux performance counters. http://user.it.uu.se/ mikpe/linux/perfctr/, 2004.

[40] S. Pingali, D. Towsley, and J. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[41] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 1997.

[42] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. May 1993.

[43] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7. Addison Wesley, 1993.

[44] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, 2002.

[45] A.T. Tai and J. F. Meyer. Performability management in distributed database systems: An adaptive concurrency control protocol. In *Fourth IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'96)*, 1996.

[46] Transaction Processing Performance Council (TPC). TPC Benchmark^{TM} C standard specification revision 5.0, February 2001.

[47] Cornell University. Testzilla - a framework for the testing of large scale distributed systems., 2006.

[48] L. Valadares, Rui Carvalho Oliveira, Isabel Hall Themido, and F. Nunes Correia. *Investigação Operacional*. McGraw-Hill, 1996.

[49] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. USENIX Association, 2002.

[50] S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. 2005.