Testing the Dependability and Performance of GCS-Based Database Replication Protocols*

A. Sousa J. Pereira L. Soares A. Correia Jr. L. Rocha R. Oliveira F. Moura
Universidade do Minho
PORTUGAL
escada@di.uminho.pt

Abstract

Database replication based on group communication, or simply GCS-based database replication, has recently been the focus of much attention as a promising technology to achieve strong consistent large-scale data management. GCS-based database replication is expected to provide increased dependability by relying on the properties of atomic multicast protocols and exhibit good performance due to the absence of distributed locking and reduced interaction among concurrent transactions.

However, until now the evaluation of such protocols has been conducted on simplistic simulation models, which fail to assess concrete implementations, or on complete system implementations which are costly to test with realistic loads and faults in a large-scale perspective.

This paper presents a hybrid model that combines simulated network and database engine components with real implementations of the replication and communication protocols. Such a model allows to precisely evaluate GCS-based database replication's overall performance and resilience when subjected to realistic loads and fault-injection campaigns in several environments.

Besides the description of the design and validation of the model, the paper presents results of the simulation of the Database State Machine replication technique using prototype implementations of its protocols.

* Research funded by FCT projects, ESCADA (POSI / 33792 / CHS / 2000) and STRONGREP (POSI / 41285 / 2001).

1 Introduction

Database replication based on group communication has recently been the subject of much attention of both theoreticians and practitioners [4, 15, 12, 24, 18]. This approach ensures consistency and increases availability by relying on the properties of atomic multicast protocols. In addition, by allowing concurrent execution of the transactions, without the need of a distributed locking mechanism, it provides good performance and scalability. In detail, the Database State Machine (DBSM) [18] works as follows. Each database site has its own concurrency control mechanism to manage the interaction between local transactions while in the execution stage. Upon receiving the commit request, a transaction enters the commiting stage and its execution outcome (i.e. write values, read-sets and write-sets) is atomically multicast [9] to all sites. Conflict among concurrent transactions is detected by a deterministic certification procedure executed at all sites. Certification uses the total ordering of the communication protocol to decide, in case of conflicts, which transactions commit or abort, ensuring consistency among all the sites.

In order to fulfill the promise of performance and reliability, one needs to realistically evaluate key components of the approach in various environments, in particular, facing a variety of fault scenarios. The DBSM approach itself has been firstly evaluated with the simulation of the certification and the communication protocols [17]. This approach allows multiple runs of the same scenario with different configuration settings, thus evaluating the impact of each parameter. On the other hand, it makes it difficult to estimate the resources used by the protocols, which compete with the database engine, as well as the reliability of the final implementations themselves. The DBSM was also evaluated by implementing it within the PostgreSQL database engine [16]. Although this provides a realistic test environment, the results are tightly related to this single database engine and a full implementation is required before initiating realistic tests. Moreover, even with a complete implementation it is costly to setup and run multiple realistic tests with slight variations of configuration parameters. This becomes particularly evident if one considers a large number of replicas and wide-area networks.

Neither of the approaches is thus adequate for evaluating early implementations of key components, as well as for testing the implementations themselves to ensure their performance and reliability. This is achieved only by submitting such components to a realistic load and fault scenario.

This paper addresses this necessity with a model of a replicated database server that combines simulation of the environment with early real implementations of key components. In particular, a real implementation of the certification and communication protocols is used, as these are the components responsible for the database replication based on the DBSM, and both the database engine and the network are simulated. This allows us, by experimenting with different configuration parameters, to assess the validity of the design and implementation decisions. In detail:

- We start by developing a centralized simulation runtime based on the standard Scalable Simulation Framework (SSF) [3], which allows the controlled execution of real implementations. This is de-

scribed in Section 2. By using the SSF, we leverage existing SSFNet [10] network simulator as a key component of a realistic environment.

- We develop a transaction processing model and a traffic generator based on the industry standard benchmark TPC-C [26]. This provides a realistic load to prototype implementations of replication components. The combined system is described in Section 3.
- We configure the simulated components to closely match a real system, namely, by instrumenting and profiling PostgreSQL [2]. This allows us to validate the model as described in Section 4.
- Finally, we apply the model to compare a centralized database with a replicated one, with various replication degrees and fault scenarios. This allows us in Section 5 to draw useful conclusions on the prototype implementation of replication protocols as well as on the DBSM approach itself.

Finally, Section 6 describes related work and Section 7 concludes the paper.

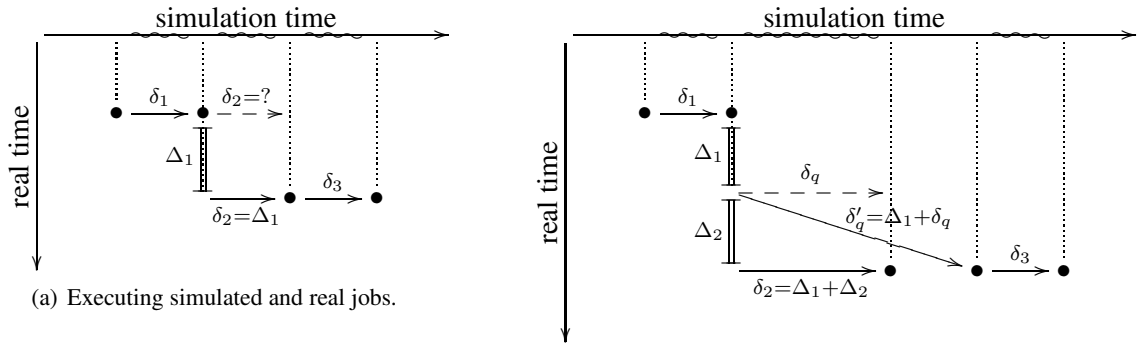
2 Simulation Kernel

In this section we briefly describe our implementation of a centralized simulation kernel [6]. This implementation is layered on top of the standard Scalable Simulation Framework (SSF) [3] thus allowing us to leverage existing models, namely, the SSFNet network simulator [10].

2.1 Scalable Simulation Framework (SSF) and SSFNet

Our simulation kernel is based on the Java version of the Scalable Simulation Framework (SSF), which provides a simple yet effective infra-structure for discrete-event simulation [3]. It comprises five base interfaces: *Entity*, *Process*, *Event*, *inChannel* and *outChannel*. An entity owns processes and channels, and holds simulation state. The communication among entities in the simulation model occurs by exchanging events through channels: An event is written to an output channel, which relays it to all connected input channels. Processes retrieve events by polling. Simulation time is updated according to delays associated with event transmission. The simulation kernel includes also a simple configuration language, that can be used to configure large models from components by instantiating concrete entities and connecting them with channels.

Simulation models are therefore built as libraries of components that can be reused. This is the case of the SSFNet [10] 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



(a) Executing simulated and real jobs.

(b) Scheduling events from real code.

Figure 1: Handling simulated and real jobs in centralized simulation.

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.

2.2 Centralized Simulation

A centralized simulation model combines real software components with simulated hardware, software and environment components to model a distributed system. It has been shown that such models can accurately reproduce the performance and dependability characteristics of real systems [6]. The centralized nature of the system allows for global observation of distributed computations with minimal intrusion as well as for control and manipulation of the experiment, namely, to perform fault injection.

The execution of the real software components is timed with a profiling timer and the result is used to mark the simulated CPU busy during the corresponding period, thus preventing other jobs, real or simulated, to be attributed simultaneously to the same CPU. In detail, a simulated CPU is obtained as follows: A boolean indicates whether the CPU is busy and a queue holds pending jobs, with their respective durations. A job with duration δ can be executed at a specific instant t by scheduling a simulation event to enqueue it at simulated time t. If the CPU is free, the job is dequeued immediately and the CPU marked as busy. A simulation event is then scheduled with delay δ to free the CPU. Further pending jobs are then considered.

Executing jobs with real code is layered on top of the same simulation mechanism. Figure 1 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 CPU is busy. 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. The CPU is marked as busy and an event is scheduled to free the CPU. 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 time-line the elapsed time spent in a real computation. Finally the second simulated job is run with duration δ_3 and the CPU marked as free afterwards, as the queue is empty.

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 1(b) 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.

2.3 Implementation Details

The implementation of the centralized simulation runtime requires a suitable clock to measure the duration of jobs as well as intercepting all calls made by real code. The profiling of real code is accomplished in the Linux 2.4 operating system using the performance counters patch [21] accessed through the Java Native Interface (JNI). This facility virtualizes CPU cycle counters for each process and thus results in a nanosecond resolution in the 1GHz Pentium III used for running simulations. The measured time can be scaled to simulate processor speeds other than that of the host processor.

Considering the restricted nature of real code we will run under control of the centralized simulation runtime, we have chosen not to intercept the standard Java runtime interface [6]. Instead, protocol code is written targeting an abstraction layer which provides job scheduling, clock access, and a simplified network interface in a single-threaded environment. The abstract interface is then implemented twice, first as a bridge to SSF and SSFNet, and then also as a bridge to the native Java API. In detail, the later uses *java.util.Timer* to schedule jobs, *java.lang.System* for clock access and *java.net.DatagramSocket* for messaging.

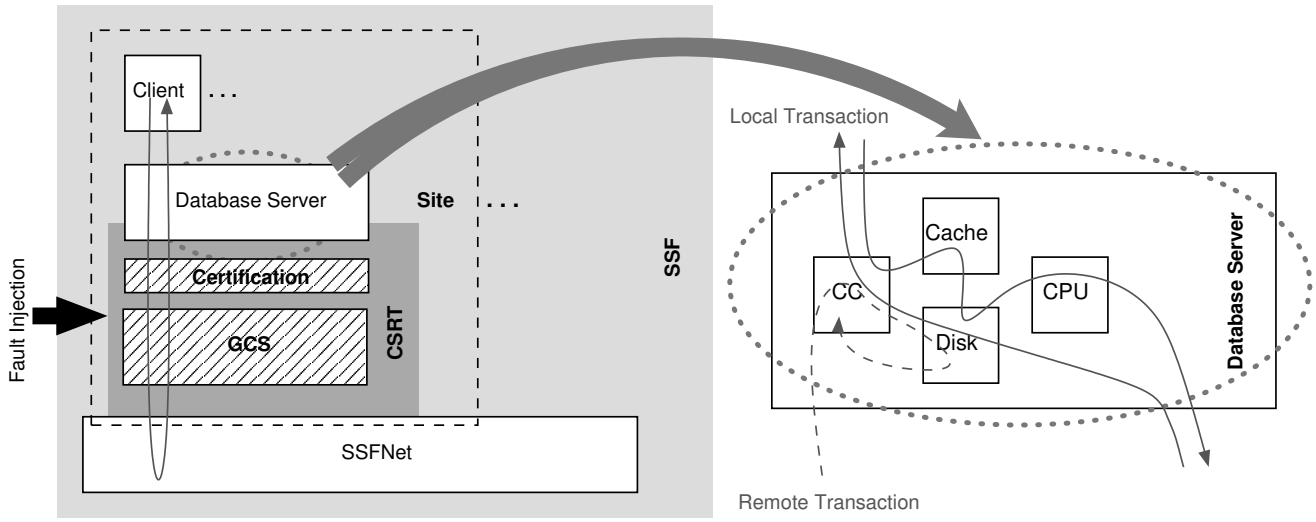


Figure 2: Overview of the replicated database model and detail of the transaction processing model.

3 Replicated Database Model

The architecture of the replicated database simulation model is presented in Figure 2. The simulation infrastructure components introduced in the previous section are depicted as shadowed boxes in the background. These are the SSF simulation kernel and the centralized simulation runtime (CSRT). Simulated components are shown as white boxes and prototypes as streaked boxes.

In short, database sites are configured as hosts in a SSFNet network and are modeled as a stack of components. At the top, each site includes a number of clients issuing transaction requests. The transaction execution path is depicted as an arrow. After being executed and when entering the commit stage, transactions are submitted to the certification layer, which uses group communication (GCS) to disseminate updates to other replicas. Upon delivery, the outcome of the certification procedure is returned to the client. The rest of this section describes in detail each of the components.

3.1 Database Server Model

The database server handles multiple clients and is modeled as a scheduler and a collection of resources, such as storage and CPUs, and a concurrency control policy. 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. Upon receiving a transaction request each operation is scheduled to execute on the corresponding resource. The execution path is shown on the right side of Figure 2. The processing time of each operation is previously obtained by profiling a real database server (Section 4 details the process).

First, operations fetching and storing items are submitted to the concurrency control module (CC). Depending on the policy being used, the execution of a transaction can be blocked between operations. The

locking policy described in this paper is based on PostgreSQL's multi-version [7]. This policy ignores fetched items, while it exclusively locks updated items. When a transaction commits, all other transactions waiting on the same locks are aborted due to write-write conflicts. If the transaction aborts, the locks are released and can be acquired by the next transaction. In addition, all locks are atomically acquired and released, also atomically, when the transaction commits or aborts, thus avoiding the need to simulate deadlock detection, which is possible as all items accessed by the transaction are known beforehand.

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. Notice that one of the simulated CPUs is also used to schedule real jobs by the centralized simulation runtime. As a consequence, transaction execution can be preempted to assign the CPU to real jobs.

A storage element is used for fetching and storing items and is defined by its latency and number of allowed concurrent requests. Each request manipulates a single storage sector, hence storage bandwidth becomes configured indirectly. A cache hit ratio determines the probability of a read request being handled instantaneously without consuming storage resources.

When a commit operation is reached, the corresponding transaction is submitted for distributed certification. This involves the identification of items read and written as well as the values of the written items. As certification is handled by real code, the representation of item identifiers and values of updated items must accurately correspond to those of real traffic. This is described in more detail in Section 3.3. When certification is concluded, the transaction is committed by finishing writing and releasing all locks held. The outcome can then be returned to the issuing client. Remotely initiated transactions must also be handled. In this case, locks are acquired before writing to disk. Local transactions holding the same locks are preempted and aborted right away, as they would abort during certification anyway.

During the simulation run, the usage and length of queues for each resource is logged and can be used to examine in detail the status of the server.

3.2 Client Model

A database client is attached to a database server and produces a stream of transaction requests. After each request is issued, the client 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 content of each request is generated according to a simulated user based on the TPC-C benchmark [26]. The database is populated according to the number of clients. It is worth noting that, our interest in TPC-C to evaluate the DBSM prototypes is just in the workload partner produced by this benchmark. Therefore, the benchmark constraints of throughput, performance, screen load and background execution of transactions are not considered here.

	Number of items			
Relations	100 (Cli.)	1000 (Cli.)	2000 (Cli.)	**Tuple size**
Warehouse	10	100	200	89 bytes
District	1×10^3	1×10^4	2×10^4	95 bytes
Customer	3×10^6	3×10^7	6×10^7	655 bytes
History	3×10^6	3×10^7	6×10^7	46 bytes
Order	3×10^6	3×10^7	6×10^7	24 bytes
New Order	9×10^5	9×10^6	18×10^6	8 bytes
Order Line	3×10^7	3×10^8	6×10^8	54 bytes
Stock	1×10^7	1×10^8	2×10^8	306 bytes
Item	1×10^5	1×10^5	1×10^5	82 bytes
Total	$\approx 5 \times 10^6$	$\approx 5 \times 10^8$	$\approx 10 \times 10^8$	

Table 1: Size of tables in TPC-C.

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: Transaction types in TPC-C.

The TPC-C benchmark proposes a wholesale supplier with a number of geographically distributed sales districts and associated warehouses as an application. An emulated client can request five different types of transactions chosen with the probability distribution presented in Table 2. This environment simulates an OLTP workload with a mixture of read-only and update intensive transactions. Namely, *Delivery* transactions are particularly CPU bound. *Payment* transactions are prone to write-write conflicts by updating a small number of data items in the *Warehouse* table . Finally, *Payment* and *Order status* execute some code conditionally, resulting in different loads. The database size is configured for each simulation run according to the number of clients, as shown in Table 1, as each warehouse supports 10 emulated clients [26].

During the run of the simulation, the client logs 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.

3.3 Certification Prototype

The distributed certification procedure [24] consists of two stages. First, just after a transaction has been executed and is ready to be certified, its associated data is gathered and atomically multicast to the group of replicas. Then, upon delivery, the second stage of the certification procedure is executed by each replica to decide whether the transaction commits or aborts.

In detail, when a transaction enters the committing stage, identifiers of read and written tuples are obtained. Our prototype assumes that each of these tuples is a 64-bit integer. The values of the written tuples are also obtained (in the simulation, tuples' sizes are used to calculate the storage usage and the amount of padding data that should be put in messages, so its size resembles the one obtained in a real system). All this information, along with the identifiers of the last transaction that has been committed locally, are marshaled into a message buffer. In practice, our protocol avoids copying the contents of buffers that are already marshaled thus improving performance.

The size of the read-set may render its multicast impractical. In this case, a threshold may be set, which defines when a table should be locked instead of a large subset of its tuples. This is similar to the common practice of upgrading individual locks on tuples to a single table lock.

Upon delivery, the message is unmarshaled. The sequence number of the last transaction committed is used to determine which transactions were executed concurrently and thus can result in conflicts. The read-set is then compared with the write-set of all concurrent transactions that have been committed. If they intersect, the transaction is aborted. Otherwise, there are no conflicts and the transaction can be committed.

Notice that this involves comparing not only individual tuple identifiers but also comparing identifiers of individual written tuples with those of the table. This is simplified by including the table identifier as the highest order bits of each tuple identifier. The runtime is minimized by keeping tuple identifiers ordered in both lists, thus requiring only a single traversal to conclude the procedure.

3.4 Group Communication Prototype

The atomic multicast protocol is implemented in two layers. A view synchronous multicast protocol and a total order protocol. The bottom layer, 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 [22] and a scalable stability detection protocol [11]. 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 [23] 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 [8, 13]. 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.

4 Model Instantiation

In this section we describe how the model is configured to reproduce the behavior of a real system. This allows us to validate the model by comparing the results of simple benchmarks run both in the simulator and in a real system. Note that the validation of the SSFNet has been done previously and is out of the scope of this paper [10].

4.1 Configuration Parameters

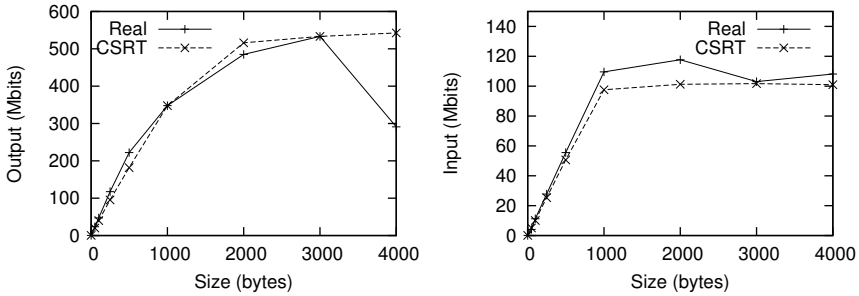
We configure our model according to the equipment used for testing. This corresponds to two servers with Pentium III 1GHz processors and with 1GB of RAM connected by a Ethernet 100 network. For storage we used a fiber-channel attached box with 4×36GB SCSI disks in a RAID-5 configuration. The file system used to hold the database (executable and data) is ext3 (Linux version 2.4.21-pre3).

The configuration of the centralized simulation runtime reduces to four parameters: fixed and variable CPU overhead when a message is sent and received. These can easily be determined with a simple network flooding benchmark with variable message sizes.

The main database server configuration issues are the CPU time and disk bandwidth consumed by each transaction. The amount of CPU consumed by the execution of each transaction is tightly related with the database system used and with the size of the database, although not significantly affected by concurrency. We therefore chose to profile PostgreSQL [2] running the TPC-C benchmark configured for up to 2000 clients but with a small number of actual running clients. As each process handles a single transaction from start to end, this reduces to profiling a process in the host operating system.

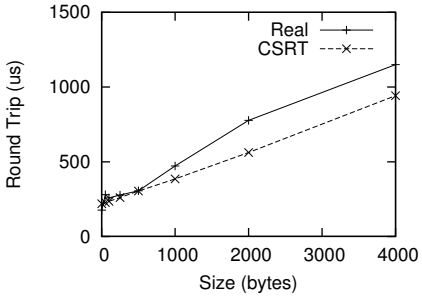
In detail, we used the CPU time-stamp counter which provides accurate measure of elapsed clock cycles. By using a virtualization of the counter for each process [21], the time elapsed when the process is not scheduled to run is not accounted for. To minimize the influence in the results, the elapsed 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



(a) Bandwidth written.

(b) Bandwidth read over Ethernet 100.



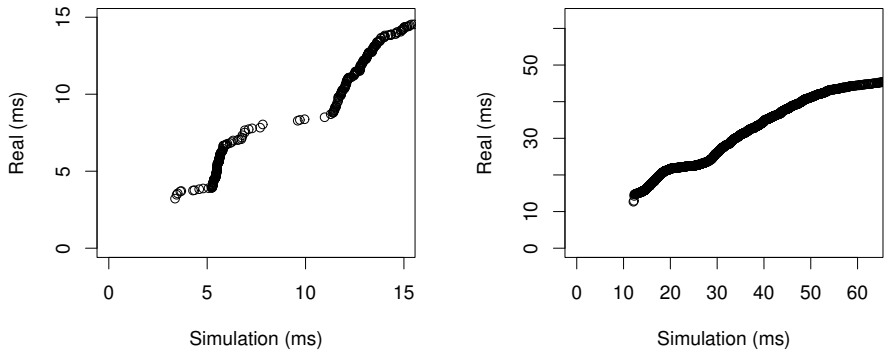
(c) Average round-trip.

Figure 3: Validation of the centralized simulation runtime.

query itself, each transaction is classified. Interestingly, the processor time consumed during commit is almost the same for all transactions (i.e., less than 2ms). In read-only transactions the real time of the commit operation equals processing time, meaning that no I/O is performed. This does not happen in transactions that update the database. The observation that the amount of I/O during processing is negligible indicates that the database is correctly configured and has a small number of cache misses.

After discarding aborted transactions and the initial 15 minutes, the resulting histogram allows an empirical distribution to be obtained and used later for simulation. However, some transaction classes (i.e. payment and orderstatus) perform some work conditionally and thus result in bimodal distributions. Therefore, we split each of these in two different classes. The resulting transaction classes can therefore be approximated by an uniform distribution.

Throughput for the storage was determined by running the IOzone disk benchmark [1] on the target system with synchronous writes of 4KB pages and a variable number of concurrent process. This resulted in a maximum throughput of 9.486MBps. As the cache hit ratio observed has always been above 98%, we configured the simulation hit ratio to 100%. This means that read items do not directly consume storage bandwidth. CPU resources are already accounted for in the CPU times as profiled in PostgreSQL.



(a) Latency of read-only transactions.

(b) Latency of update transactions.

Figure 4: Validation of the DBSM model.

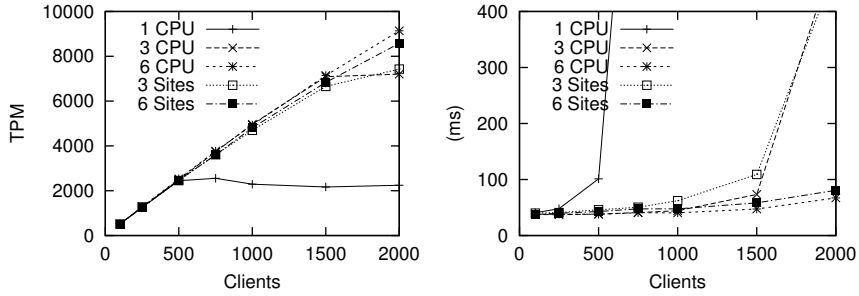
4.2 Validation

The 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 3(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. Notice that crossing the 4KB virtual memory page boundary severely impacts performance in the real system. Figure 3(b) shows the result of the same benchmark at the receiver, limited by the network bandwidth. Finally, Figure 3(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 architecture of the simulated database server has been used before [5]. Validation is thus performed to ensure that the current configuration closely matches a specific real system. Unfortunately, we cannot run the benchmark in a real setting with thousands of clients as is possible with the simulation. Instead, we have used a run of the TPC-C benchmark with 20 clients only. Quantile-quantile plots (Q-Q plot) of observed latencies is presented in Figure 4, showing that the resulting distributions match.

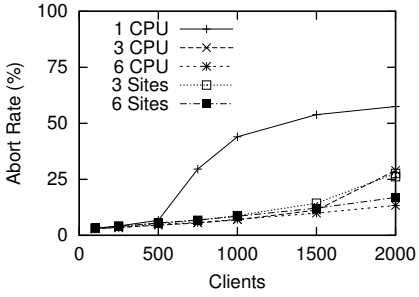
5 Experimental Results

In this section we use the simulation model to evaluate the performance and dependability of the prototype implementation of key components of a replicated database. In detail, we start by presenting performance results of two different systems. These results are then explained by an in-depth look into measurements of resource usage. Finally, we present results obtaining when injecting faults in one of the system models.



(a) Throughput.

(b) Latency.



(c) Abort rate.

Figure 5: Performance results.

5.1 Performance

Figure 5 compares the performance of replicated databases with 3 and 6 sites in a local area network. Each site is configured as previously described in Section 4 with a single CPU. Clients are evenly distributed across all sites. For comparison, we present also results obtained with a single site configured with 1, 3 and 6 CPUs. This is useful as a baseline for comparison to assess the scalability of the replication protocol. For each of these scenarios, we run simulations with an increasing number of clients from 100 to 2000 and compute the resulting latency, throughput and abort rate.

Figure 5(a) presents the number of committed transactions per minute (tpm). Aborted transactions are not resubmitted. It can be observed that the system with 3 sites scales gracefully up to about 1500 clients and 7000 tpm. With 6 sites the system scales past 2000 clients and 9000 tpm. In fact, the performance of each distributed system is very close to the centralized system with the same number of CPUs. The saturation of systems with 3 CPUs, both centralized and distributed, can also be observed in latency in Figure 5(b) and in the resulting abort rate in Figure 5(c). Notice that the 1 CPU system handles little more than 2000 tpm with 500 clients.

	500 Clients	1000 Clients		1500 Clients	
Transaction	1 CPU	3 CPUs	3 Sites	6 CPUs	6 Sites
delivery	1.30	0.94	1.62	1.14	3.70
neworder	1.56	0.88	1.59	1.27	1.46
payment (long)	15.81	16.69	21.05	23.87	28.36
payment (short)	10.21	11.36	14.30	17.83	22.18
orderstatus (long)	5.02	7.98	7.86	6.29	8.30
orderstatus (short)	0.00	0.00	0.00	0.00	0.00
stocklevel	0.00	0.00	0.00	0.00	0.00
All	6.73	6.99	8.84	9.94	12.12

Table 3: Abort rates (%).

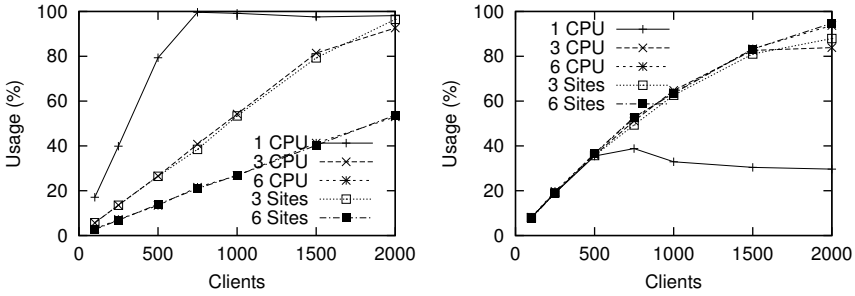
5.2 Resource Usage

These results suggest that available CPUs are the limiting resource and thus that the DBSM approach to replication is a viable alternative to symmetrical multi-processor machines. However, one should look closer into the small increase in the number of aborted transactions, as this might impact differently the various transaction types. The breakdown of the abort rate is presented in Table 3. The only statistically relevant are the *NewOrder* and *Payment* transactions (each account for 44% of the transactions) and it shows that replication impacts only the later. This is explained by the fact that the *Payment* transaction is highly prone to write-write conflicts by updating the small *Warehouse* table, which results also in a high abort rate even in centralized settings. The increased concurrency due to lack of distributed locking thus results in some additional opportunities for conflicts. This issue should be dealt preferably at the application level, by preventing conflicts in the same manner as would be necessary for a centralized multi-version database server.

The information logged by the simulation runtime allows the precise evaluation of resource usage. Figure 6(a) presents average usage of each of the involved CPUs, both by simulated transaction processing jobs and by protocol jobs. This justifies the limitation of throughput and the latency shown in the previous section. In detail, it shows that a single CPU is a bottleneck with only 500 clients, approaching 100% utilization. Note that the single host with 3 CPUs reaches the same saturation point close to 1500 clients, which is 3 times the load at which the single CPU does. Notice that with 6 CPUs and 6 replicas, the system can perfectly handle the amount of clients simulated, showing linear performance.

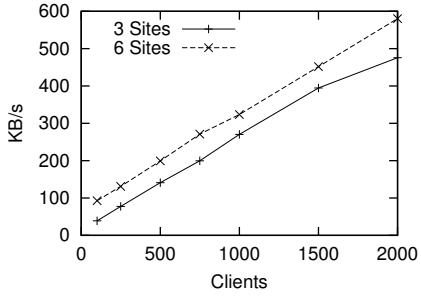
Figure 6(c) shows that with 6 CPUs, regardless of being centralized or in 6 different sites, the bottleneck is disk bandwidth. This a direct consequence of using a read one/write all total replication technique and it is a limitation to the scalability of DBSM systems. The problem can be mitigated by using partial replication [24], while still providing the increased resilience from replication.

Figure 6(c) shows a linear increase in transmitted bytes with the number of clients and the transaction



(a) CPU.

(b) Storage.



(c) Network.

Figure 6: Resource usage.

throughput. This shows that the protocol is scalable in terms of bytes transmitted. The difference shown between the 3 sites and the 6 sites models is due to the additional group maintenance traffic generated by the additional participants. The amount of traffic generated shows that a typical local area network is adequate to handle the traffic of a replicated database with thousands of clients and even that it is realistic to consider using the technique for distant database sites connected by a wide area network.

5.3 Fault Injection

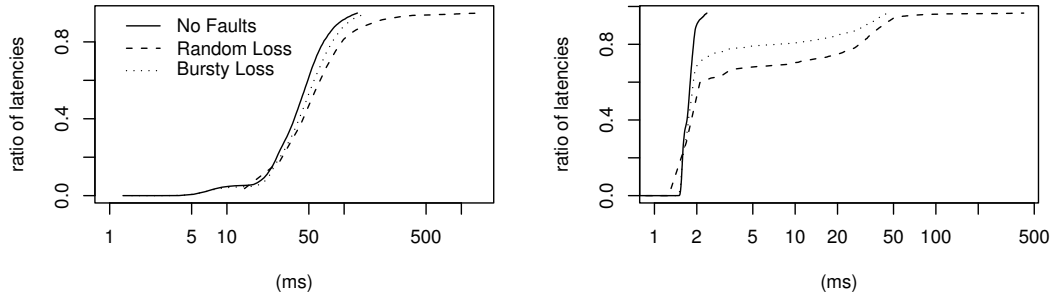
In this section we present the results obtained when injecting various fault types in the simulation models described in the previous sections. Faults can be injected by the centralized simulation runtime by intercepting calls in and out of the runtime as well as by manipulating its state. The targets are therefore the certification and group communication protocols.

The evaluation of the results is two-fold: First, we 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 4 and ensures the safety of the approach and of the prototype implementation in maintaining consistency.

Second, we are interested in the impact of faults in the performance of the system. Besides crashes, which as expected have a profound impact in performance by disconnecting a number of clients, the types of faults causing more performance degradation are those causing message losses. Figure 7(a) plots the empirical

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 4: Types of faults injected.



(a) Transaction latency.

(b) Certification latency.

Figure 7: Performance results with fault injection.

cumulative distribution functions (ECDF) of transaction processing latency measured in runs with 3 sites and 750 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 than before. Table 5 shows also an increase in CPU usage by real, showing the extra work by the protocol in retransmitting messages.

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 6.

This is explained by Figure 7(b), which shows the latency of certification alone. It can be observed that the long tail with random loss is directly caused by delays in certification. In fact, this tail corresponds to the group protocol blocking a few times for short periods during the simulation run. Blocking is caused by

Run	Usage
No Faults	1.22
Random Loss	1.90
Bursty Loss	1.89

Table 5: Protocol CPU usage (%).

	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 (long)	15.43	25.94	18.74
payment (short)	10.85	18.96	12.10
orderstatus (long)	5.43	5.93	4.32
orderstatus (short)	0.00	0.00	0.00
stocklevel	0.00	0.00	0.00
All	6.72	11.94	7.96

Table 6: Abort rates with 3 sites and 1000 clients (%).

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 buffer space or by allocating a dedicated sequencer process. In the future, it should be solved by avoiding the centralized sequencer.

Note also in Figure 7(b) 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 [19] is necessary for efficient deployment in wide area networks.

6 Related Work

The proposed simulation model can easily be extended to evaluate several modifications of the DBSM approach. Namely, an alternative proposal [16] avoids multicasting read-sets which can be reasonably large in some kinds of transactions. The trade-off in this approach is two-fold: it reduces the resilience of the protocol as it confines the certification capability to one of the replicas and also increases latency since an additional communication step is required to multicast the outcome of the certification to all replicas. With the aim at improving the protocol scalability, an approach exploiting *partial replication* appeared in [24]. In a partial replication scenario, data is fragmented across the replicas, thus a coordinated certification process is required. This is achieved extending the DBSM protocol with a final agreement step. Such step is responsible for the agreement of all replicas in the outcome of the transaction.

Simulation has previously been used to evaluate DBSM replication [17]. The high level of that model easily allows the experimentation of several variations of the basic DBSM approach. Namely, of the reordering technique before certification. Nevertheless, the simplicity of the traffic generator used severely limits the detail of the results regarding conflicts.

An implementation of the DBSM using the PostgreSQL database engine is available and has been previously evaluated [16]. Although such an implementation provides results for a real running system, it is much harder to evaluate. For instance, it would be very difficult to set up and run the same experiments presented in this paper. This possibility is also invaluable when optimizing and debugging certification and communication protocols, as one can generate different environment scenarios to stress the implementation [6]. The usage of a high-level simulation model for the database engine allows us to easily experiment with different concurrency control models. In fact, although we have not presented it in this paper, we have already implemented different locking policies and are evaluating its impact in DBSM replication. This would be very hard to do using PostgreSQL.

The use of simulation models has been used frequently to evaluate the performance of database processing techniques. In fact, our model of database processing is close to that of [5]. Our work differs mostly in the configuration of the model according to a real database engine and the consequent ability to validate the results experimentally as well as on the integration of real code. Combining simulated and real components in a single model has been described previously in the context of fault-tolerant distributed systems [6]. By using a standard simulation API we are however able to reuse an existing simulation model, SSFNet [10].

We have decided to use a high level model of the CPU time consumption by transaction processing, in particular, by modeling it after the results of profiling. If we were interested in studying this parameter in more detail, we could extend the proposed simulation model with a low level model of access to items as has been previously described [14].

7 Conclusions

This paper addresses the performance and dependability evaluation of complex and large-scale distributed systems by proposing a model that combines simulated components of the environment with early implementations of key components. In detail, we evaluate the Database State Machine (DBSM) approach to database replication. The results presented in this paper confirm the the usefulness of the approach, by illustrating the scalability to 3 and 6 sites with minimal replication overhead. The results presented are also able to pinpoint a limitation of the current implementation of the group communication protocol in a scenario with random network loss and its impact in the quality of service provided to end-users of the replicated database.

We are currently applying the system to evaluate several design decisions, namely, the usage of partial replication [24] and various techniques to achieve it in wide area networks, optimistic total order protocols [25] and semantic reliability [20]. The resulting system has also been put to use for automated regression tests. As different components are modified by separate developers, the ability to autonomously run a set of realistic load and fault scenarios and automatically check for performance or reliability regressions has proved invaluable.

References

[1] IOzone filesystem benchmark. http://www.iozone.org, 2004.

[2] PostgreSQL. http://www.postgresql.org, 2004.

[3] Scalable simulation framework. http://www.ssfnet.org, 2004.

[4] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases (extended abstract). In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–172. ACM Press, 1997.

[5] R. Agrawal, M. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems (TODS)*, 12(4):609–654, 1987.

[6] G. Alvarez and F. Cristian. Applying simulation to the design and performance evaluation of fault-tolerant systems. In *IEEE International Symposium on Reliable Distributed Systems*, 1997.

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

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

[9] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4), December 2001.

[10] J. Cowie, H. Liu, J. Liu, D. Nicol, and Andy Ogielski. Towards realistic million-node internet simulation. In *Proc. of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, Nevada, Jun 1999.

[11] K. Guo. *Scalable Message Stability Detection Protocols*. PhD thesis, Cornell University, Computer Science Department, May 1998.

[12] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group multicast. In *IEEE International Symposium on Fault-Tolerant Computing*.

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

[14] W. Keezer. Array-driven simulation of real databases. In *Proc. of the 1998 Winter Simulation Conference*, Washington, DC, 1998.

[15] B. Kemme and G. Alonso. A suite of database replication protocols based on communication primitives. In *IEEE International Conference on Distributed Computing Systems*, 1998.

[16] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 134–143. Morgan Kaufmann, 2000.

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

[18] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 2003.

[19] F. Pedone and A. Schiper. Generic broadcast. In *International Symposium on Distributed Computing (DISC)*, September 1999.

[20] 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.

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

[22] 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.

[23] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *IEEE International Conference on Distributed Computing Systems*, May 1993.

[24] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *IEEE Int'l Symp. Networking Computing and Applications*. IEEE CS, Oct 2001.

[25] 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*, pages 190–199. IEEE CS, October 2002.

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