Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado em Engenharia Informática
2º Semestre, 2007/2008

Efficient Middleware for Database Replication

André Abecasis Gomes Ferreira, nº26836

Orientador
Prof. Nuno Manuel Ribeiro Preguiça

31 de Julho de 2008

Nº do aluno: 26836
Nome: André Abecasis Gomes Ferreira

Título da dissertação:
Efficient Middleware for Database Replication

Palavras-Chave:

- Replicação
- Execução especulativa
- Bases de Dados
- Middleware

Keywords:

- Replication
- Speculative execution
- Databases
- Middleware

Resumo

Os sistemas de bases de dados são usados para armazenar informação nas mais variadas aplicações, incluindo aplicações Web, empresariais, de investigação e pessoais.

Dada a sua larga utilização em sistemas fundamentais para os utilizadores, é necessário que os sistemas de bases de dados sejam eficientes e fiáveis. Adicionalmente, para que estes sistemas possam servir um elevado número de utilizadores, é importante que as bases de dados sejam escaláveis, podendo processar grandes quantidades de dados. Para tal é necessário recorrer à replicação dos dados. Num sistema replicado, os vários nós contêm cópias da base de dados. Assim, para garantir a convergência das réplicas, as operações de escrita têm de ser efectuadas sobre todas as réplicas. O modo como esta propagação é efectuada dá origem a duas estratégias diferentes. A primeira, em que os dados são propagados assincronamente após a conclusão de uma transacção de escrita, conhecida como replicação assíncrona ou optimista. A segunda, em que os dados são propagados sincronamente durante a transacção, conhecida como replicação síncrona ou pessimista.

Na replicação pessimista, ao contrário da replicação optimista, as réplicas mantêm-se consistentes. Assim, esta aproximação permite simplificar a programação das aplicações, porque a replicação dos dados é transparente para as mesmas. No entanto, este tipo de aproximação apresenta problemas de escalabilidade, devido ao número de mensagens trocadas na sincronização, que obriga a um atraso na conclusão da transacção. Assim, o utilizador tende a experimentar uma latência bastante superior na abordagem pessimista.

Neste trabalho apresenta-se o desenho e implementação dum sistema de replicação de bases de dados, com semântica snapshot isolation, usando uma aproximação de replicação síncrona. O sistema é composto por uma réplica primária e um conjunto de

réplicas secundárias que replicam totalmente a base de dados. A réplica primária executa as transacções com operações de escrita, enquanto que as restantes executam as transacções que só têm operações de leitura. Após a conclusão de uma transacção de escrita na réplica primária as alterações são propagadas para as restantes réplicas. Esta aproximação adequa-se a um modelo de utilização em que a fracção de operações de leitura é bastante superior à de operações de escrita, podendo a carga das leituras ser dividia pelas várias réplicas.

Para melhorar o desempenho do sistema, os clientes executam algumas operações de forma especulativa, de modo a evitar que fiquem em espera durante a execução de uma operação na base de dados. Deste modo, o cliente pode continuar a sua execução enquanto a operação é executada na base de dados. Caso o resultado devolvido ao cliente se verifique ser incorrecto, a transacção será abortada, garantindo a correcção da execução das transacções.

Abstract

Database systems are used to store data on the most varied applications, like Web applications, enterprise applications, scientific research, or even personal applications.

Given the large use of database in fundamental systems for the users, it is necessary that database systems are efficient e reliable. Additionally, in order for these systems to serve a large number of users, databases must be scalable, to be able to process large numbers of transactions. To achieve this, it is necessary to resort to data replication. In a replicated system, all nodes contain a copy of the database. Then, to guarantee that replicas converge, write operations must be executed on all replicas. The way updates are propagated leads to two different replication strategies. The first is known as asynchronous or optimistic replication, and the updates are propagated asynchronously after the conclusion of an update transaction. The second is known as synchronous or pessimistic replication, where the updates are broadcasted synchronously during the transaction.

In pessimistic replication, contrary to the optimistic replication, the replicas remain consistent. This approach simplifies the programming of the applications, since the replication of the data is transparent to the applications. However, this approach presents scalability issues, caused by the number of exchanged messages during synchronization, which forces a delay to the termination of the transaction. This leads the user to experience a much higher latency in the pessimistic approach.

On this work is presented the design and implementation of a database replication system, with snapshot isolation semantics, using a synchronous replication approach. The system is composed by a primary replica and a set of secondary replicas that fully replicate the database- The primary replica executes the read-write transactions, while the remaining replicas execute the read-only transactions. After the conclusion of a

read-write transaction on the primary replica the updates are propagated to the remaining replicas. This approach is proper to a model where the fraction of read operations is considerably higher than the write operations, allowing the reads load to be distributed over the multiple replicas.

To improve the performance of the system, the clients execute some operations speculatively, in order to avoid waiting during the execution of a database operation. Thus, the client may continue its execution while the operation is executed on the database. If the result replied to the client if found to be incorrect, the transaction will be aborted, ensuring the correctness of the execution of the transactions.

Index

Index of Figures

Index of Tables

1 Introduction

1.1 Motivation and context of the problem

A large number of different applications are built on top of database systems. These applications include most sites with dynamic contents, corporate applications (e.g. banking, airline reservations, etc.), scientific and even personal applications. Thus, database systems need to provide a service that is reliable and efficient, in order to be useful and practical to users. Additionally a database system must be scalable to process large amounts of operations and data.

To guarantee reliable transaction execution, database systems implement the ACID properties (Atomicity, Consistency, Isolation and Durability). In a system that provides Atomicity, a transaction must be performed atomically, so that all of the transaction operations are performed and if one is not performed then none must be performed. The Consistency property allows the database to be inconsistent while the transaction is being processed, but before the transaction begins and after the transaction terminates the database must be consistent. To reach Isolation, the order for the execution of the transactions operations must be serializable, meaning that independently of the schedule of the transaction history, the outcome must be equal as if the transactions where executed sequentially with no overlapping. In most database systems, Isolation is relaxed to improve performance, thus allowing some limited concurrency. Finally, Durability guarantees that if the transaction commits, and the user has received its result, the modifications to the database are not lost, even if the system crashes.

Replication is a mechanism used to improve availability and performance of database systems, as well as to allow the scalability of the database. When replication is used, several sites, or replicas, maintain copies of the database. Under full replication the

whole database is stored at every replica. Under partial replication, only a subset of the data is stored at each replica. Although, partial replication is very attractive, since a smaller amount of resorts is necessary, it is harder to implement automatically, as the data needed for each operation may be distributed over several sites.

When using replication, when an update transaction is executed, all replicas must be updated to maintain consistency. The propagation of updates may be postponed to the end of the transaction or it may be executed immediately for each write operation.

Deferred update replication techniques offer advantages over immediate update techniques, such as better performance, simplified server recovery and a lower deadlock rate. However, there is a drawback to this approach: the lack of synchronization during transaction execution that may increase greatly the transaction abort rates.

There are two major replication approaches, synchronous or eager replication, and asynchronous or lazy replication. When a transaction is executed in a replica the changes must be applied to all other replicas. In eager replication, the updates are propagated inside the transaction, i.e., before finishing the transaction. On commit, all replicas must be updated synchronously using an atomic commit protocol. In lazy replication the updates are propagated asynchronously only after the commit of the transaction, i.e., after the transaction has committed locally, therefore allowing the replicas to diverge. Each approach has its own advantages and drawbacks.

Eager replication provides serializable execution since there are no concurrency anomalies, but there is a decrease in the global performance since extra messages are exchanged as a consequence of the synchronization to achieve consistency between the replicas. The synchronization itself is a problem for efficiency since before returning the result of the transaction to the client, all replicas must be updated. Therefore clients experience a large latency using eager replication systems. This approach provides single-copy consistency, meaning that clients observe the system as if it has only one copy of the data, allowing the programmer to use a familiar model.

Lazy replication allows more efficient implementations as replicas do not need to be synchronously updated inside a transaction. However, as this approach allows a

transaction to see old values, the replicas can diverge. To solve the divergence, a reconciliation mechanism is necessary. If the number of nodes scales up, the number of reconciliations tends to scale up, which may lead to system delusion (the database replicas will be inconsistent) [9]. In lazy replication the principle of single-copy consistency is not observed. In fact, the programmers must be aware of this drawback and adapt their applications accordingly.

Synchronous replication with good performance is still an area of research, as recent work show [4, 5, 6, 7, 8]. Commercial databases often use asynchronous replication that tolerates inconsistencies among replicas.

1.2 Solution proposed

In this work, we have tried to develop an efficient middleware for database replication. Since a lazy replication requires reconciliation because it allows replicas to diverge, we have chosen to implement a synchronous approach. Our approach provides single-copy consistency, making it much easier to implement new applications because the programmer observes one (logical) copy of the data.

The architecture of the system is loosely based on the Ganymed [5] system, as it is a replication system based on primary-copy. In this approach, updates are first committed on a primary copy of the data, and then applied to the other copies, thus guaranteeing that replicas evolve to the same state. A read operation may execute in any of the system copies, thus allowing distributing the load among system replicas.

The proposed middleware system is composed by clients and a set of replica servers. Users' applications run in clients and access the replicated database system using a custom JDBC driver. Thus, applications can use our replication middleware without any modification.

At each replica server, the middleware component uses a JDBC driver to communicate with the local database copy. The local database fully replicates the data managed by

the system. The database itself was not modified. At any moment, there is a designated master/primary replica.

The system provides Snapshot Isolation, an isolation level that relaxes serializability. When a transaction is started, the system creates a (logical) snapshot of the database with the currently committed values. The snapshot is used to read values and write new values, inside the transaction. Transactions do not observe updated values from transactions that commit after the snapshot. Snapshot Isolation can be implemented using a lock-based approach or the *first-committer-wins* rule to prevent lost-updates. In this case, when a transaction tries to commit, if it has updated data that was also updated by a transaction that committed after the start of this transaction, this transaction must be aborted. In the former case, the locking subsystem prevents concurrent updates from being executed.

Unlike Ganymed, we avoid requiring applications to declare transactions as read-only, while using a similar strategy for load balancing read-only and update transactions. For this end, the following approach is used. When a client starts a transaction, a transaction is started both on a replica and on the master, guaranteeing that the same database snapshot is used. After this, the operations of the transaction are sent only to the replica. If the transaction ends without any write operations, the transaction is committed in the replica and it is discarded in the master. If one of the transaction operations happens to be a write operation, then the transaction is discarded in the slave replica and the transaction in the master continues the execution from the write operation and thereafter. In practice, this is similar to the Ganymed functionality, which implements a Primary-Copy approach, where read-only transactions execute on slave replicas and update transactions execute on the master replica. The difference is that read-only transactions do not need to be declared a-priori.

One disadvantage of a Primary-Copy is that it can become a bottleneck of the system, particularly with a large number of replicas, since all update transactions are executed on the primary, as well as being a single point of failure. For addressing failures, we have implemented a failure recovery mechanism that replaces the primary in case of failures. Regarding load, as it is usual that the fraction of update transactions be small,

we expect that this primary-copy approach does not limit the system's performance, in practice.

When an update transaction is committed, the updates must be propagated to the slave replicas. The master bundles the updates in a write set which it propagates to the slave replicas.

In our approach, as read-only transactions are executed in slaves, it is necessary to balance the load among the replicas. Additionally, it is necessary to guarantee that an update transaction from a client does not observe a snapshot that does not include updates observed/executed by previous transactions from the same client. Unlike Ganymed, we implement our approach without a central scheduler to provide these properties. For balancing the load, transactions are executed on randomly chosen replicas. The system guarantees that a client does not observe snapshots of the database older than the previously seen snapshot.

Our system resorts on speculative execution to improve the global performance. It also decreases the number of messages exchanged by grouping specific messages and propagating them in a set.

Regarding speculative execution, it is used on the clients, along with the regular execution. When a client requests an operation prone to be speculative, a speculative execution is started. Without waiting for the database server reply, the client assumes the most plausible reply and continues executing (speculatively). When the reply finally arrives at the client, the reply is compared with the result assumed before. If both are equal, then the client continues the execution. On the other hand, if the server result differs from the result returned to the application, the speculative execution is invalid. To signal this to the application, the transaction will fail and the application code must re-execute the transaction, as usual.

In summary, the goal of this dissertation was to design and implement an efficient middleware for database replication that provides good performance with strong consistency. To attain this goal, speculative execution was used on the clients, along with some optimizations that are later explained on the implementation chapter.

1.3 Main contributions

This project contributes with the design and implementation of a middleware database replication system. As already discussed, a replication system aims to improve availability and performance of the database system.

Speculative execution is also applied to the replicated system, in order to explore the advantages of this type of execution, namely to improve the efficiency of the system by using it in the clients. The clients are allowed to carry on with the execution, instead of blocking in remote operations to the database.

The overall performance of the developed system was evaluated with a standard realistic benchmark, the TPC-C. TPC-C is an on-line transaction processing (OLPT) benchmark. It is composed by a set of five different concurrent transactions that simulate a computing environment, where users execute transactions on a database. TPC-C reflects the transactions executed at warehouses, like entering and delivering orders, checking the status of these orders, recording payments, and monitoring the stocks.

1.4 Outline

This dissertation is organized as follows: Chapter 2 describes the related work and it is divided in three sections. Section 2.1 introduces the speculative execution and related works. Section 2.2 presents the different database isolation levels. Section 2.3 discusses database replication and various works related, ending with a comparison between specific properties of the systems. Chapter 3 presents the design of the system and Chapter 4 discusses the implementation of this design. Chapter 5 presents the evaluation of the system. Finally Chapter 6 presents the conclusions of the thesis.

2 Related work

In this chapter, we will present works related to ours, organized in three sections. The first section addresses speculative execution. The second section briefly presents isolation levels used in database systems. The third section addresses database replication, focusing on synchronous replication approaches.

2.1 Speculative Execution

Speculative execution is the execution of some operations that might not be necessary. In general, speculative execution is used as a performance optimization. For example, modern pipelined processors resort to speculative execution to optimize conditional branches instructions by assuming the most probable branch decision and executing from there immediately, instead of waiting for the decision. Later, if the correct value is different, the execution past the branch decision is discarded (i.e. the pipeline is emptied).

Speculator [1] allows clients to execute operations at the application level speculatively, performing client level rollback if needed. Speculator extends the Linux kernel to support speculative execution, preventing processes from exteriorizing output until their depending speculations are verified as correct and assuring no speculative state is directly seen by non-speculative processes. In [1] the system was used to improve the performance of distributed file systems.

Speculator was used to explore speculative execution in a client-server database system [2]. In a database system, a client waits on a remote call to the database for the results of submitted operations. In this system, when a client submits operations to the database server the client speculatively assumes a result and continues executing. Later, the

assumed result is compared with the one from the server. If they are equal, the speculative execution is made definite. On the contrary, if they are different, the speculative execution is rolled back and the program is re-executed with the correct result.

Zyzzyva uses speculation in a state machine replication system that tolerates byzantine failures. This system serves replicas speculatively, executing operations before they can definitely establish the final execution order. If the order turns out to be incorrect, the system restores a previous version.

2.1.1 "Speculative Execution in a Distributed File System" [1]

Motivation and objectives

Distributed file systems performance is worse than that of local file systems because clients must contact remote servers to obtain data or, at least, to verify that local copies are up-to-date. Even distributed systems with loose consistency and safety, where the number of synchronous messages exchanged is greatly reduced, are outperformed by local file systems.

This work adds support for speculative execution to the Linux operative system. Then it uses the mechanism to increase the performance of distributed file systems with no loss in consistency, by allowing clients to speculatively execute using the cached values before their consistency is confirmed.

Architecture and functionality

To implement the Speculator the authors were forced to do some changes to the Linux kernel. The system forbids a speculative process to externalize output, since a non-speculative process cannot see the state of a speculative process. This lead to a set of modifications to prevent a process from externalizing output, like altering operative system mechanisms like fork, exit, signals, pipes, fifos, sockets, etc. The output is buffered to be later externalized when the speculative execution becomes definitive.

The checkpoint of the speculation is performed doing a fork of the process, but the child process is not put on the run queue, it is just used to store the process state. Later if the speculation execution is to be definitive this child process is discarded, if the speculation is to be rolled back then the child process assumes the identity of the current process.

A distributed file system using Speculator maintains basically the same architecture of a distributed file system without Speculator, that is, a set of clients sends requests to the distributed file system server(s). But using Speculator involves modifications to the client, the server and the network protocol. The clients do caching of the file system. In the event of a read operation from the client application, the cached values are used immediately, initiating a speculation, without waiting for the server reply. When the reply arrives at the client, the speculation is committed if the value is the same as the value used in the speculation, or rolled back if the value is different. In the case of speculative write operations the file server checks if the speculation hypothesis is true or false since it knows the true state of the file system. If it is false the write operation fails and the speculation fails.

2.1.2 "Execução especulativa em bases de dados" [2]

Motivation and objectives

This project was developed with the future idea of using the information obtain with its execution to build an efficient middleware replicated system recurring to speculation, since replicated database systems have a substantial overhead caused by the data replication, which diminishes its efficiency. Usually that is the trade-off for choosing replication, a higher security of the data but lower efficiency.

The objective of the project was to test the application of speculative execution to a database system in order to test the improvement in its efficiency.

Architecture and functionality

The system is composed by any number of clients and proxies. The clients submit operations to the database using a special JDBC driver to communicate with the proxy.

The proxy receives operations submitted by the client and uses a JDBC driver to forward them to the database server.

The Speculator [1] library is used to allow clients to continue execution speculatively instead of waiting for remote operations to the database server. In this process, the client checkpoints the current state and continues execution with the most probable reply from the database server. If the guessed reply proves to be correct, the client saves precious execution time avoiding blocking during remote operations. If the server reply is different from the guessed reply then the client rollbacks to the checkpoint and re-executes with the correct reply this time.

While executing, the JDBC driver starts a new speculation when an operation prone to be speculative is executed (if not already in speculative mode). Operations for which the result cannot be guessed are not prone to speculation obviously.

For implementing out system, some changes were introduced to the speculator. First, as the Speculator interface was built to be used in supervisor mode, new system calls had to be added to the kernel to execute the Speculator functionalities in user mode. Second, since the output is postponed in speculative mode, some changes were made to allow communication between proxies and clients. The reason for this is because a non-speculative process (e.g. the proxy) cannot see the state of a speculative process (e.g. the client).

Kaffe, an open-source Java Virtual Machine with user level threads, was used because Speculator only supports single-threaded programs. The database system used was the open-source system PostgreSQL.

2.1.3 “Zyzzyva: Speculative Byzantine Fault Tolerance” [3]

Motivation and objectives

Byzantine fault tolerance pretends to provide support against byzantine failures in a distributed system, which are caused by erroneously components that lead to arbitrary faults. In a byzantine fault tolerant system the correctly functioning components reach consensus regardless of the byzantine faulty components.

Byzantine fault tolerant (BFT) replication is increasingly attractive for practical deployment because the hardware is becoming inexpensive, therefore lowering the costs of implementing BFT. Additionally, several improvements have been proposed to BFT replication techniques, thus diminishing the overhead involved. Nevertheless, the BFT replication algorithm still imposes a considerable overhead.

This work proposes the use of speculation to improve a Byzantine fault tolerant state machine replication using tentative speculative execution on the server to reduce the latency experienced by clients.

Architecture and functionality

The system is composed by a finite number of clients and 3f + 1 replicas. Up to f nodes can experience byzantine failures.

The Zyzzyva protocol proceeds in a sequence of view runs. In each view, there is a designated primary replica.

The protocol has three sub-protocols: 1- agreement, which orders requests for the replicas to execute; 2- view change, which coordinates the election of a new primary replica; 3- checkpoint, to limit the state that is stored by replicas, reducing also the cost of view changes.

In the agreement protocol, the client sends its requests to the primary, which forwards the requests to the replicas that speculatively process the request and reply to the client. The client assumes a request is completed when it receives 3f + 1 mutually-consistent replies. When the client receives between 2f + 1 and 3f replies, it gathers 2f + 1 replies in a commit certificate that it sends to the replicas, waiting for 2f + 1 replies to consider the request complete. If less than 2f + 1 replies are received, the client resends the request to all replicas which may need to contact the primary. The client detects a faulty primary if the ordering of the messages is inconsistent, and warns the replicas, leading to a view change to elect a new primary.

The view change protocol is used to elect a new primary. A new primary is to be elected when a replica takes knowledge of its faulty state, or because f +1 replicas

agreed to a new primary. This protocol has a phase called “I hate the primary”, which serves to prevent a replica to abandon a view unless it is guaranteed that all replicas will do the same, leading to the election of a new primary. When a replica suspects the primary is faulty, the replica sends a message to all replicas warning them of the suspicion. If the replica receives f + 1 answers confirming the suspicion then it sends another message to every replica asking for a view change, containing the information that f + 1 replicas confirmed this suspicion, and stops sending messages in this view. A replica that receives this last message automatically commits to the view exchange, so eventually all replicas will commit to the view change.

For every fixed number of requests received by a replica, a checkpoint is made. Each checkpoint is associated with a snapshot of the database. The replica maintains a stable checkpoint and can store a tentative checkpoint. When a tentative checkpoint is generated is one replica, a message is sent to every other replica. When f + 1 replicas answer to the message the checkpoint is committed and the history before the committed checkpoint is garbage collected.

The protocol has speculative execution because replicas execute requests before the total order is established. In each replica there’s a committed history and a speculative history. The committed history is the history from the last committed checkpoint till the last request of the max commit certificate, which is the certificate that covers the largest stored history. The speculative history is the history thereafter, i.e. it includes the operations for which the server has not established that the execution order is correct. If the execution order is determined to be incorrect, e.g. due to a malicious primary node, the speculative history is discarded and the replica restarts executing operations immediately after the committed history.

2.2 Database Isolation Levels

Database systems isolate the transactions from each other to prevent the inconsistency of the data in a multi-user environment. As full serializability restricts concurrency, weaker isolation levels have been defined. The ANSI SQL-92 [13] standard defines

isolation levels relying on the following phenomena (or anomalies) that can occur during concurrent transaction execution:

1- Dirty Read – occurs when a transaction reads uncommitted data that is later rolled back;

2- Non-Repeatable Read – occurs when a transaction executes the same query multiple times and the results are different because they were modified by an interleaving transaction.

3- Phantom – occurs when transaction T1 executes a query that returns a set of data from the database and another transaction creates data that satisfy the previous transaction query. If the transaction T1 then executes the same query the set obtained will be different.

The standard SQL [13] defines the following four isolation levels that partially avoid these phenomena:

1- Read Uncommitted – This is the lower isolation level and does not prevent any of the phenomena.

2- Read committed – Prevents only the Dirty Read phenomenon.

3- Repeatable Read – Prevents both the Dirty Read and the Non-Repeatable Read phenomena.

4- Serializable – The highest isolation level. The Serializable isolation level prevents all the above phenomena.

The transactional system used in our middleware (PostgreSQL) provides the four ANSI SQL isolation levels but internally only two distinct isolation levels exist, which correspond to the ANSI SQL levels Read Committed and Serializable. Read Uncommitted is Read Committed internally and Repeatable Read is Serializable. This is possible by the standard SQL since the isolation levels translate internally to equal or higher isolation levels, therefore preventing the same phenomena as the isolation level requested by the user, or more.

Snapshot Isolation is a level of isolation not defined by the standard SQL that avoids the same phenomena as defined in the ANSI SQL Serializable isolation level, but does not provide full serializability. The advantage of snapshot isolation is that it is more efficient since the read-only transactions never block nor abort. In this isolation level, a transaction executes in a snapshot of the database, which is taken before the transaction begins and that contains all the committed values this far. This snapshot is used by the transaction to read and update values and is only committed to the database when the transaction commits. Two concurrent transactions conflict with each other if they both write the same data items. In this case, one of the transaction must abort.

The PostgreSQL system Serializable isolation level corresponds to the Snapshot Isolation. This may seem awkward but in practice it is possible to obtain an execution with Snapshot Isolation equivalent to an execution with the ANSI SQL Serializable isolation level.

2.3 Database Replication

Replication is used to improve reliability, fault-tolerance and availability. In database systems, when using replication, database copies are stored in multiple nodes (replicas).

If a master replica exists, it processes all the requests: we call this approach a primary-backup scheme. In contrast in a multi-master scheme any replica can process a new request and distribute the new updates to the other replicas. This later approach requires a distributed concurrency control and a solution for handling conflicting transactions. Eager replication solves the conflicts between transactions by preventing the conflicts. The conflict is detected before the committing of the transaction and one of the conflicting transactions is aborted. Lazy replication allows transactions to commit and resolves the conflicts after the commit. Eager solutions promote consistency by propagating changes to replicas within the transaction boundaries. Lazy solutions promote efficiency over consistency by only propagating changes to replicas after the transaction commits (with a possible considerable delay). We start by presenting an

example of a system that uses lazy replication [9], and continue with systems that use eager replication, as this late approach is the focus of our work

2.3.1 "The dangers of replication and a solution" [9]

Motivation and objectives

Scalability is an important issue for database systems. This work studies the scalability problems associated with update anywhere-anytime-anyway transactional replication system on mobile environments. The results show that the deadlocks and reconciliation rates raise greatly, leading to unstable behavior.

In this study it is also shown that the use of primary copy approaches reduce the aforementioned problems and it is proposed a novel algorithm that relies on the primary copy approach, while providing disconnected operations and serializable transaction execution.

An analysis of different replication strategies (for mobile computing)

Eager replication reduces update performance and increases transaction response times because it adds extra updates and messages to the transaction, since all replicas are updated inside the transaction. Additionally, it does not support mobile applications, since most nodes are usually disconnected.

Lazy replication is a solution for mobile applications since the updates can be propagated to other nodes asynchronously, outside transactions. However, in this approach, data on replicas can become stale, allowing a transaction to observe old committed values, increasing the probability of conflicting transactions. With a lazy strategy, when some replica updates have already been committed when a conflicting transaction is detected, it is necessary to rely on a reconciliation mechanism.

Simple lazy replication only works well with low loads and few nodes. If the application scales up to a larger number of nodes, if the nodes are disconnected more often, or if the delays of messages are longer, then it is necessary to reconcile

conflicting transactions more often. As reconciliation fails, the nodes begin to diverge to the point where the database becomes inconsistent with no obvious way to repair.

In eager replication, locking detects possible conflicts before they occur converting them to waits or deadlocks. The waits cause delays while the deadlocks create application faults. In lazy replication the transactions that would wait using an eager strategy face reconciliation, which frequency is determined by the much more frequent waits using a lazy strategy.

Due to the need of contacting the master, lazy-master replication is not an option for mobile applications. In this type of replication, objects have owners that store their current "official" value. Updates are executed in the master first, and then they are propagated to the other replicas. Lazy-master replication is slightly less deadlock prone than eager replication but it requires contact with object masters, which store the object's current value.

Architecture and functionality

The solution proposed is based on a two-tier replication approach, which is a modified mastered replication scheme. Each object is mastered by a node, to avoid reconciliation. This scheme assumes two kinds of nodes: mobile nodes and base nodes. The mobile nodes are disconnected most of the time. They store a replica of the database and may originate tentative transactions. They may also master some of the data items. Base nodes are always connected and also store a replica of the database. This nodes master most of the items. The items have two versions at mobile nodes: a master version, which is the most recent value received from the master; and a tentative version, which is an updated version of the local objects modified by a tentative transaction. Following the same idea there are two kinds of transactions: base transactions that work only on master data and produce new master data; and tentative transactions that work on local tentative data and produce new tentative versions and a base transaction to be run on the base nodes.

Mobile nodes accumulate tentative transactions that run against the tentative database stored at the node. These tentative transactions are reprocessed as base transactions

when the mobile node reconnects to the base nodes, but may fail when being reprocessed.

2.3.2 "A suite of database replication protocols based on group communication primitives" [4]

Motivation and Objectives

Replication in most commercial database systems is based on optimistic replication. This approach allows inconsistencies and relies in centralized algorithms. The reason for this is the strong belief of database designers that synchronous and update-everywhere approaches, based on 1-copy-serializability, have performance and scalability problems.

The goal of this work was to design synchronous, update everywhere protocols, based on group communication, that do not suffer from performance and scalability problems.

Architecture and Functionality

The system is a distributed database consisting in a fixed number of nodes. The database is fully replicated, so each node has a full copy of the database. The nodes themselves, communicate via message passing. The group communication subsystem provides two different services, a basic service, with no ordering guarantees; and a total order service, where all the sites deliver the messages by the same total order. Regarding message delivery, there is an atomic delivery and a non-atomic delivery. A node that fails is excluded from the group by group maintenance services. The members of the new group coordinate the delivery of the failing node pending messages.

The system uses a version of the all available copies approach. Reads operations are executed locally. Write operations are deferred until all read operations have been executed and are broadcasted in a write set. Since it is an all available copies approach, an update request must be performed by all available replicas. The ordering of

conflicting transactions is established by the total order service from the communication model.

There are four levels of isolation supported by the system: Serializability, Cursor Stability, Snapshot Isolation and a Hybrid Protocol. Snapshot Isolation (SI) is implemented using the protocol presented in figure 1.

The transaction manager of each node N coordinates the operation requests of the transactions as follows:

(1) *Local Reading Phase:* For each read operation $r_i(X)$, reconstruct the version of X labeled with T_j where T_j is the transaction with the highest $TS_j(EOT)$ so that $TS_j(EOT) \leq TS_i(BOT)$.

(2) *Send Phase:* If T_i is read-only, then commit. Else bundle all writes in WS_i and multicast it (total order service). The WS_i message also contains the $TS_i(BOT)$ timestamp.

(3) *Lock and Version Check Phase:* Upon delivery of WS_i, perform in an atomic step:
For each operation $w_i(X)$ in WS_i:
   (a) If there is no write lock on X and the current version of X is labeled with T_j. Then, if $TS_j(EOT) > TS_i(BOT)$, stop checking locks and abort T_i. Otherwise grant the lock.
   (b) If there is a write lock on X or a write lock is waiting. Then let T_j be the last transaction to modify X before T_i: if $TS_j(EOT) > TS_i(BOT)$, then stop checking locks and abort T_i. Otherwise enqueue the lock request.

(4) *Write Phase:* Whenever a write lock is granted, perform the corresponding operation.

(5) *Commit Phase:* Whenever all operations have been executed, commit T_i and release all locks.

Figure 1: Replication protocol guaranteeing snapshot isolation (from [4])

For implementation of Snapshot Isolation each object in the system has various versions. An older version can be reconstructed applying undo operations to the object until the required version is generated. Each one of these versions is labeled with the transaction that created the version. The *first writer wins* strategy is used, so when a transaction T wants to write to an object that was updated by other transaction after the transaction T started, the transaction T is aborted.

The beginning (BOT) and the ending (EOT) of a transaction are identified by timestamps. Since the system is distributed, the timestamps (TS) need to be

synchronized. The system uses the write sets sequence numbers as a global virtual time, to synchronize the timestamps.

The decision of the abort or commit of a transaction is local to each node, so no messages are sent between the nodes.

The protocol, illustrated in figure 1, is composed by five phases. The first phase is the reading phase, where the read operations are executed in a snapshot of the database. Phase two is the send phase, where the write sets are sent to every replica using a total order service unless the transaction is read-only and therefore is immediately committed. The next phase, executed on the delivery of the write sets, is the lock phase which includes a version check. If a write operation from transaction T (bundled in the write set), updates an object that was updated by another transaction that committed after the start of T, T is aborted. The same happens for conflicts with transactions that are in the process of committing, but are ordered before T (in the timestamp order). To avoid the overhead in the case of frequent aborts a preliminary check can be done locally, before sending the write set, and the transaction is aborted and restarted if a conflict is detected. This does not avoid the need for the remote check. Finally, the protocol ends with the write phase and the commit phase respectively. The write phase performs the write operations when the write lock is granted and the commit phase commits the transaction when all operations from the write set have been executed and releases all the locks for this transaction on the replica.

The system also supports three other protocols, for which that we now briefly summarize: Serializability is implemented using strict 2-phase locking (2PL); Cursor Stability, is implemented using short read locks, and may lead to non-repeatable reads because a lock is placed on an item as long as an SQL cursor is positioned on the item, but is released when the cursor moves on; and the Hybrid Protocol, which is a mixture of the 2PL protocol, used for update transactions, and the SI protocol, for read-only transactions, which must be pre-declared.

The broadcast primitives are relaxed to minimize message and logging overhead, which leads to three versions of the Serializability, Cursor Stability and Snapshot Isolation

protocols. The versions of the protocols are non-blocking, blocking and reconciliation based.

In the non-blocking versions of the protocols, the delivery of the write set and the commit message is atomic, and the abort message does not need to be sent atomically. This way, each node can decide to abort in-doubt transactions of a failed node, without contacting other nodes. A transaction is seen as in-doubt by a node N, when it is invoked at a node N' and the write set is delivered to node N, but the abort or commit message has not. With this approach, a group change that excludes a failing node is not announced to the application of a node N, until all messages the failing node might have delivered are delivered at node N.

In a blocking version of the protocol, some of the overhead of the non-blocking approach is avoided by allowing not reaching a decision about the transactions of the failed node. The delivery of the write set in this approach is atomic, but the commit message is not. In this version of the protocols, a node can no longer decide independently on an in-doubt transaction, because the delivery of the commit and abort messages are non-atomic, which allows other nodes, including the failed one, to deliver the commit or abort and terminate the transaction. So, a coordination protocol is needed in this version. When a transaction is in-doubt at all nodes, it must be blocked until the recovery of the failing node. If there is a node where the transaction is not in-doubt, that node must inform the nodes where the transaction is in doubt.

Finally, the reconciliation based version that does not broadcast any message atomically and is an alternative when performance needs to be improved. In this case, a failing node may have committed a transaction but the other nodes may have not received the write set or may have decided to abort the transaction. When the failing node recovers, it must conciliate its database with the databases of the working nodes and compensate the changes done by the transaction.

The best protocol and version depends on the workload and the system configuration.

2.3.3 "Ganymed: Scalable Replication for Transactional Web Applications" [5]

Motivation and objectives

Replication is a common solution for the scalability requirements of database services like data grids and large scale web application. However to implement replication it is necessary to balance trade-offs between consistency and scalability. Commercial systems usually choose scalability, giving up on consistency. Other common solutions offer limited scalability in exchange for consistency.

Ganymed is a database middleware that intends to provide scalability in a replicated database system without sacrificing consistency. The system targets the needs of typical dynamic content generation with a large amount of complex reads and a small number of short update transactions.

Architecture and functionality

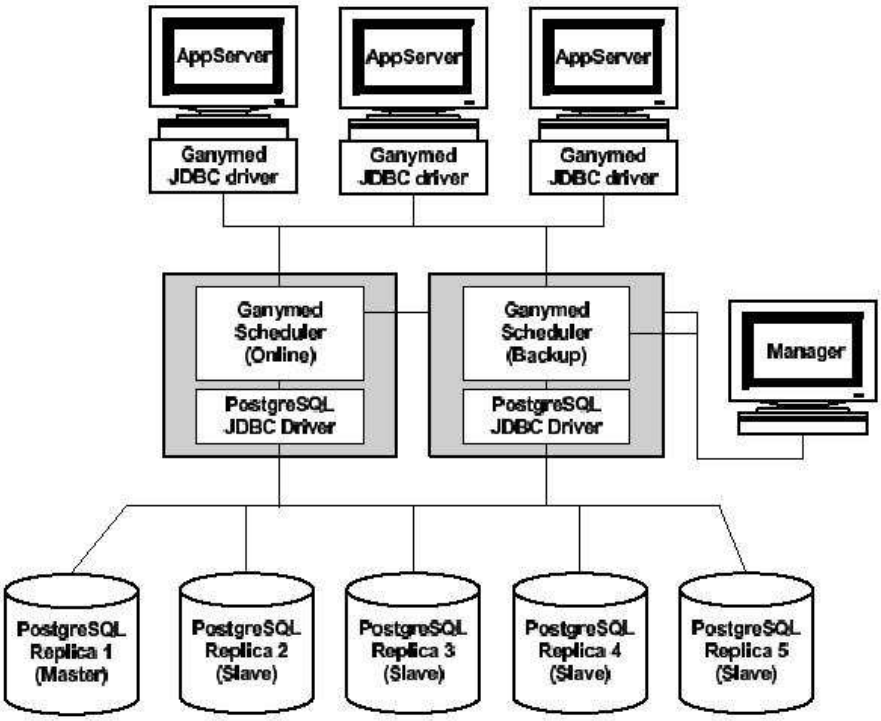

Figure 2: Ganymed Prototype Architecture (from [5])

The Ganymed architecture, depicted in figure 2, is composed by four components: the clients, the replicas, the scheduler and a manager component.

The clients are application servers that connect to the scheduler through a custom JDBC driver, for submitting operations. For the clients the scheduler abstracts a single database with Snapshot Isolation. The scheduler communicates with the replicas using a PostgreSQL JDBC driver. The set of replicas contains a Master replica and several Slave replicas. Each replica must be a database providing Snapshot Isolation. The replicas can be added and removed from the system at runtime and the Master role can be assigned dynamically. There is also a manager component which runs in a dedicated machine and monitors the system. This manager component is used to allow the user to configure the system, adding and removing replicas, and to substitute a failing scheduler.

The main component is the middleware scheduler that controls the execution of client transactions over the set of replicas. There is a backup scheduler to substitute a failing one.

The scheduler implements the RSI-PC (Replicated Snapshot Isolation with Primary Copy) algorithm. The algorithm separates update and read-only transactions. Updates go directly to the master replica without delay and queries go to the slave replicas.

After committing a transaction in the master, the updates are bundled into a write set, which is broadcasted to all replicas. For the extraction of the write sets, the JDBC interface was extended with the necessary logic to collect changes to the database. The write sets are table row based.

Read-only transactions need to call Connection.setReadonly(), so they are known in advance. Read-only transactions that are known in advance are assigned to a slave replica according to the *least pending requests first* rule, even if the master has capacity to process them. If the read-only transactions are not known in advance, or there are no slave replicas present, the transactions have to be processed by the master replica.

Read-only transactions will always see the latest snapshot of the database. If the replica does not have the latest version of the database the execution of the transaction is suspended until all needed write sets have been applied to the replica, which is verified by the scheduler. To avoid this delay the client may specify the allowed staleness of the transaction, or send the read-only transactions to the master replica.

If a replica fails, the system uses the remaining replicas. If the master fails, a slave replica becomes the master. The reaction to failing replicas can be done by the scheduler without intervention from the manager console.

Ganymed can be used of-the-shell for any application, as it does not impose any special data organization, structuring of the load, or particular arrangement of the schema. The only requirement for achieving good performance is for read-only transactions to be known in advance.

2.3.4 “Database Replication Using Generalized Snapshot Isolation” [6]

Motivation and objectives

Conventional snapshot isolation requires transactions to observe the latest snapshot of the database, which is not suitable for replicated databases, because, unlike centralized databases, the latest snapshot is not available in every replica – in fact, to determine which is the latest snapshot, it might be necessary to contact all replicas.

This work extends snapshot isolation to suite replicated databases, by defining Generalized Snapshot Isolation, which allows the transactions to observe older snapshots of the database, instead of just the latest. A particular case of the Generalized Snapshot Isolation is the Prefix-Consistent Snapshot Isolation, where the snapshot contains, at least, the writes of previously committed transactions.

Generalized Snapshot Isolation

Generalized snapshot isolation (GSI) is an extension of the conventional snapshot isolation, maintaining many of its properties. This approach allows a transaction to execute in a snapshot older than the latest, instead of just the latest like the conventional snapshot isolation. This increases the probability of aborts for update transactions in consequence of the increase in the number of “concurrent” transactions.

Replicas can process read-only transactions locally as no conflicts may arise. An update transaction may be executed locally, except the commit, which requires certification to detect write-write conflicts. A transaction reads only committed data and does not commit if updates conflict with another committed update transaction ("first-committer-wins"). If there is any intersection of the write set of the transaction, with the write sets of the update transactions that have committed after the snapshot used by the transaction, the transaction aborts; otherwise it commits.

Prefix-Consistent Snapshot Isolation (PCSI) is an instance of the GSI that uses the most recent snapshot available, by including the updates of all transactions that are in the transaction's workflow and have committed before the transaction started.

Architecture and functionality

The system architecture is composed by any number of clients, any number of replicas running Snapshot Isolation and a certifier. Transactions are executed on the replicas. If the transaction is read-only then it executes locally in the replica and does not involve the certifier, but the commits of update transactions must be certified.

The authors propose two algorithms to implement PCSI. The algorithms vary regarding the certifier implementation.

The first algorithm has a centralized certifier which also stores the write sets of the transactions and is implemented on a master replica. The remaining replicas work as slaves that communicate only with the master. Read and write operations are locally executed on the slave replicas and read-only transactions do not communicate with the master, while update transactions must be certified, therefore must communicate with the master.

The other algorithm has a distributed certification: all replicas execute transactions and certify update transactions. For the delivery of the write sets for certification, for all replicas, an atomic broadcast is used. Thus the ordering of transactions is established and all replicas may reach the same decision on which replicas commit/abort.

In both algorithms the only operation that requires remote communication is the certification of an update transaction. Upon the commit of the update transaction, the write set is applied at the other replicas, according to the algorithm.

Regarding failures, a recovering site must apply the effects of all messages which it has not delivered or that it has not processed. In the first algorithm, when a site recovers it reads the last committed snapshot locally and asks for the most recent updates from the master. In the second algorithm, the recovering site also reads the last committed

snapshot locally but then broadcasts a recovery message to ask for the missing updates, which may be sent by any replica.

2.3.5 "Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases" [7]

Motivation and objectives

In replicated databases, an important feature to achieve good performance is the load-balancing of the requests among running replicas. Conventional load-balancing strategies, like round robin and least active connections, have a good load balance but introduce memory contention (as transactions are placed without any knowledge on which data they will access). LARD (Locally-Aware Request Distribution) reduces memory contention using a content aware load-balancing technique, and it is very effective for read-only static content Web workloads – small files – but it can work poorly for workloads where requests with large working sets are frequent, like database transaction workloads, besides handling updates inefficiently.

This work introduces a memory-ware load balancing (MALB) technique, which uses knowledge about the size and the contents of the working set of transactions to assign them, so that they can execute in main memory, reducing disk I/O.

Architecture and functionality

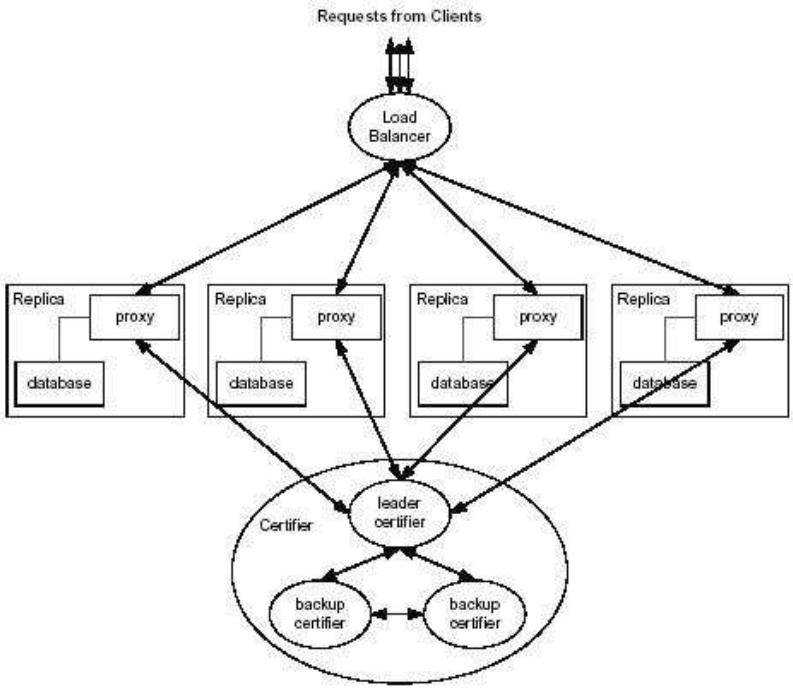

Figure 3: Tashkent replicated database design (from [7])

Tashkent+ is composed by the database replicas, a certifier and a load balancer to accept requests from clients and dispatch them to the set of replicas. As the previous system, Tashkent+ uses generalized snapshot isolation (GSI). Replicas communicate only with the certifier. The certifier is used to verify which concurrently executed update transaction can commit, by deciding the commit order and checking for conflicts. Each replica has a proxy attached that intercepts the requests transparently. The proxies use an algorithm to prevent bursts from overloading the database. When a commit is attempted by an update transaction, the proxy sends a request to the certifier, to certify the write set, so this one can detect write-write conflicts. The successfully certified write sets are recorded in a persistent log. Data consistency is maintained propagating the write sets to all replicas.

For handling failures, a primary-backup scheme is used. There is a backup load balancer. When the primary fails, clients use the backup and all active transactions are aborted and retried. The certifier is also replicated for availability, with one leader certifier and two backups.

The main innovation of Tashkent is the MALB algorithm that intends to dispatch transactions to replicas avoiding memory contention. To this end, the state of each database replicas is monitored by taking estimates of the working sets. This is achieved exploring information in the transactions execution plans, which contain the tables and indices used and how they are accessed. The estimates are used by the MALB algorithm to create transaction groups so that each one fits in the main memory of a replica and to dynamically allocate replicas for the transaction groups.

The load balancer receives replica load information on the CPU and the disk I/O channel utilization from lightweight daemons running in each replica. The loads of each transaction group are compared and additional replicas are allocated to the most loaded group.

In a stable workload, the partitioning of the transaction groups across the replicas can be made permanent by the load balancer. Each replica receives a subset of the transaction types, so the tables not used at the replica can be dropped or allowed to become out-of-date. So, updates to these unused tables can be filtered since they do not have to be processed by the replica. This technique is called update filtering, and can be enabled on the Load Balancer. Dynamic replica allocation is disabled when update filtering is enabled.

2.3.6 "Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication" [8]

Motivation and objectives

Update everywhere replication has several limitations, as high deadlock rates, message overhead and poor response times.

The goal of this work is to circumvent these limitations by using shadow copies to first execute the transactions locally, thus postponing the propagation of the updates. The system relies on group communication to pre-order the transactions and to acquire all locks needed by a transaction in an atomic step.

The proposed solution is integrated into the PostgreSQL database system.

Architecture and functionality

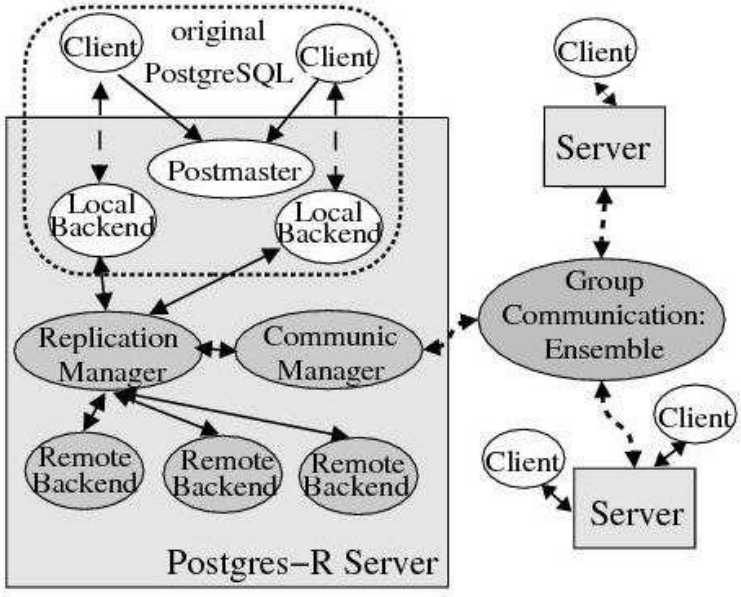

Figure 4: Postgres-R Architecture (from [8])

Postgres-R was implemented as an extension to PostgreSQL, maintaining the same functionality but providing replication as an additional feature.

The Postgres-R architecture consists of several nodes (servers), each running an instance of Postgres-R. The database is replicated at all sites. When clients want to access the database they send a request to a postmaster process running in a replica. The postmaster handles the transactions required by the client.

Update transactions execute on shadow. The changes to the shadow copies are propagated to the other sites at commit time, thus reducing the message. A transaction can read its previous writes by reading the shadow copies. Constraints can be checked and triggers fired during the update on the shadow copy.

Read transactions are executed locally. A read transaction is aborted when a conflicting write arrives. This approach avoids deadlocks and inconsistent executions.

The replication protocol has an eager approach, and executes a transaction in four phases: I - Local Read Phase, II - Send Phase, III - Lock Phase, IV - Write Phase. In the first phase, all read operations are performed locally and the write operations are executed on the shadow copies. In the second phase, if the transaction is read-only, it commits locally. Otherwise, the write sets are broadcasted to all sites. The third phase is used to request all write locks needed by the transaction, thus guaranteeing that the transaction can be serialized in respect to concurrent transactions. Finally, on the fourth phase, the updates of the transaction are executed and after the commit all locks it required are released.

To provide serializability, Postgres-R uses a reliable total order group communication primitive to multicast the write set and to determine the serialization order of the transactions. A local site commits a transaction whenever the global serialization order has been determined. It does not have to wait for the other sites to have executed the transaction because it relies on the fact that the other sites will serialize the transaction in the same way.

Since PostgreSQL uses locking at the table level, which is not desirable for efficiency reasons, Postgres-R uses a simple (logical) tuple level locking scheme based on key values. This is implemented resorting to the shadow copies. Since during the Read Phase, the updates of a transaction are executed on the shadow copy, locally, the updated key values can be include on the write set that will be propagated to the other sites. This allows a logical tuple level locking during the Lock Phase.

2.3.7 “The Database State Machine Approach” [10]

Motivation and objectives

As many of the previous works, the motivation for this work is the need for good performance in replicated databases that provide 1-copy serializability.

The Database State Machine approach for synchronous database replication proposes an approach to deal with replicated databases over a cluster of servers, which differs from traditional mechanisms by not using distributed transactional mechanisms. The approach relies on deferred updates, but is meant to reduce the transaction abort rate using a reordering certification test to look for possible serializable executions before deciding to abort.

Architecture and functionality

The system is composed by a set of replicas, each with a full copy of the database, and a set of clients. The sites communicate via message passing and fail independently by crash (Byzantine failures are not supported). Clients submit transactions that are executed by the database sites. The system provides 1-copy serializability for the transaction execution.

The transactions are locally executed with no interaction with other sites and locally synchronized using strict two phase locking. The system has a termination protocol, which is executed when the client requests the commit of a transaction. In this protocol, the transaction's write set and read set are atomically propagated to all replicas, where the transaction is certified and committed, if possible. The certification is used to ensure one-copy serializability, aborting a transaction if its commit leads the database to an inconsistent state.

Each replica behaves like a state machine, so when processing the delivered transactions, all replicas should reach the same state. To achieve this, all certifiers must enforce that write-conflicting transactions are applied in the same order, by granting their locks by the same order as they are delivered. To certify the delivered transaction, the certifier checks if the write sets of committed transactions conflict with the read set of the committing transaction.

Read-only transactions are locally executed and committed, and do not need certification, but may be aborted due to local deadlocks or during the certification of remote update transactions. To avoid aborting read-only transactions during certification

of remote update transactions, the transactions must be declared as read-only transactions. Then, remote update transactions wait for the conclusion of the read-only transaction, to obtain the locks needed.

The system presents a modification to the certification test that lowers the abort rates. The idea is to reorder the transactions that are trying to commit, to increase the probability of committing them, by constructing a serial order that lowers the possible conflicts.

On recovery from failure, a replica receives the updates of the missed transactions (while the replica was down) by communicating with a replica that has seen all transactions in the system.

2.3.8 "Revisiting the Database State Machine Approach" [11]

Motivation and objectives

The goal of this work is to extend the DSMA to avoid the extraction and propagation of read sets, while incurring in no communication overhead and guaranteeing the serializability of transactions.

Architecture and functionality

The base of DSMA* remains the same as the DSMA: a non centralized approach with deferred update replication, where read-only transactions are processed locally, and update transactions do not require synchronization between replicas until commit time. Each site has a full-copy of the database and transactions are locally executed according to strict two-phase locking (2PL).

The DSMA* extends DSMA to avoid the need of read sets during certification. The extraction of read sets implies modifying the database internally, or to parse the SQL statements outside of the database. Instead of using the read set for certification, the write set is used. By using just the write sets to check for conflicts, only snapshot isolation is achieved. So to achieve serializability the authors divide the database in

logical sets. For each set, corresponding dummy rows are created. Each time a transaction reads or writes to an object of the set, a write operation is executed in the corresponding dummy rows. This strategy is used to detect serializability conflicts. For example, a transaction T1 reads from a record R from a set S, and a transaction T2 writes to the record R and commits. When T1 tries to commit there will be a writing conflict, therefore T1 is aborted. But this approach has a drawback. The transaction T1 may want to read a row from the record R that is different from the row written by T2. Nevertheless, it is still aborted. So to achieve serializability, the number of possible conflicts rises greatly.

As in DSMA sites communicate with each other through atomic broadcast. Each site has the role of a replica manager and all replicas receive and process the same sequence of requests in the same order.

2.3.9 "Gorda: An Open Architecture for Database Replication (extended)" [12]

Motivation and objectives

Database vendors provide no support for third party replication. Thus, to provide replication, two approaches are possible. The first one is to modify the database. This approach, that is only possible when the source code is available, is hard to maintain when new versions are released, and it restricts portability. The second approach is to implement a middleware layer that uses the database. This approach introduces a performance overhead due do additional communication.

This work intends to address this problem, allowing the implementation of third party replication in existing database systems.

The solution proposed is a novel architecture (GORDA) along with a programming interface (GAPI) for replication, allowing the implementation of different strategies on any compliant DBMS. This approach intends to be cost-effective and efficient, by enabling the reuse of replication protocols or components in multiple DBMSs and allowing close coupling with DBMS internals.

Architecture and functionality

The base of the architecture is a reflective system that allows a computation to be inspected or manipulated. For example, in an object oriented system the invocations of methods on objects can be reflected as objects, which can be manipulated and inspected. These reflected objects are called the meta-objects, in opposition to the ordinary, or base, objects.

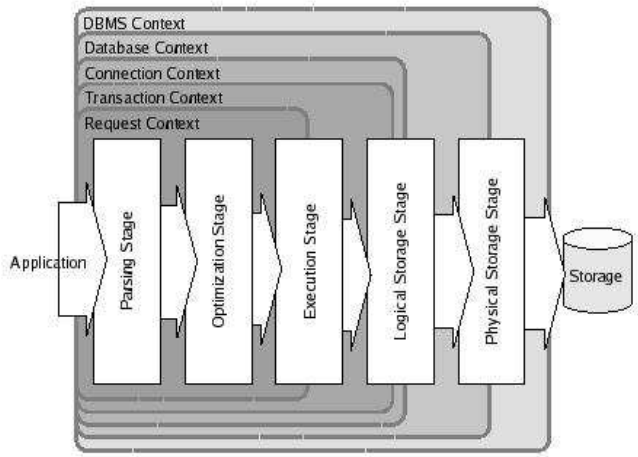

Figure 5: Processing stages and contexts (from [12])

The transaction processing is abstracted as a pipeline, as depicted in figure 5. The pipeline is composed by the following stages: Parsing, Optimization, Execution, Logical Storage and Physical Storage. The idea is to issue notifications at the meta-level when proceeding from one stage to another.

In the database system, the following meta-objects are used: the DBMS and Database, which expose metadata and allow notifications of lifecycle events; the Connection, which reflect existing client connections to the database; the Transaction, to notify

events related to a transaction; and the Request, to ease the manipulation of requests within a connection to a database and the corresponding transactions.

These meta-objects and pipeline stages were chosen to obtain an adequate level of granularity. If a very fine granularity was used, the interface could not be easily mapped to different DBMSs and the performance overhead would be very high. With a very large granularity, like the one obtained when wrapping the server and intercepting the client requests as they are being issued, the interface exposes too little information to be useful.

In [12], the authors describe some examples of how to implement Primary-Backup, State-Machine replication and Certification-Based replication protocols using the proposed interface. But the interface can be used to implement any type of replication protocols with its possible multiple variants. For example, the certification based protocol presented provides a variant with snapshot isolation, based on a distributed certification process (as used, for example in GSI [6]).

To implement primary-backup, the Transaction context is used to capture the moment when a transaction starts to execute and commits, or rollbacks, at the primary; and the Execution Stage is used to provide the object set needed to extract the write set of a transaction from the primary and propagate it to the backup replicas. In a synchronous approach for the Primary-Backup, there are six steps. In an asynchronous approach the fourth and fifth steps are postponed. In the first step the client sends the requests to the primary. On the following step, when the transaction begins, the primary is notified to register information about this event. The third step happens after the processing of an SQL statement and it is used to retrieve the updates, which are stored along with previous updates from this transaction. The fourth step is used when the primary is about to commit a transaction, allowing it to broadcast the stored updates to all replicas. The fifth step is to allow the primary to commit the transaction, after the write set is received and executed at all replicas. Finally, the sixth step is for the primary to reply to the client.

The certification-based protocol execution is similar to the primary-backup, as only the certification is distributed. Therefore, it also uses the Transaction context and the Execution Stage. It is also composed of six steps and the first four are the same as for the primary-backup implementation. On the fifth step, after receiving the write set, each replica certifies the transaction, deciding to commit or abort it. All replicas reach the same decision since the certification is deterministic and an atomic broadcast establishes a global total order. In case of abort, the replica that received the updates from the client cancels the commit, through the context component, and the other replicas discard it. In case of commit this replica allows the transaction to continue and the other replicas are able to execute the updates. The sixth and final step is for the replica contacted by the client to reply him.

2.3.10 Summary

Table 1 presents a comparison of the presented systems, according to six specific properties.

The group communication column indicates whether the system relies on a group communication subsystem to exchange information among replicas, or not. In the later case, the replicas usually do not communicate among themselves.

In the replica consistency (client view) column, we classify the systems on whether the client can observe divergent replicas or not (despite the fact that, internally, replicas may become divergent for short periods of time).

The update propagation column presents the strategy to propagate updates to the replicas. This can be done using an eager approach or a lazy approach.

The architecture column presents the system architecture. Some systems have fully distributed architectures, while others include centralized components (e.g. primary-copy).

The isolation column shows the highest level of isolation provided by the system.

Finally, the last column addresses the technique to detect conflicting transactions. This can either be done by the underlying database system or by the middleware. Some systems have a specific entity to detect and resolve the conflicts, called the certifier.

From the table, we see that different alternatives exist for each aspect and that they can be combined in different ways. But most of the systems presented use similar approaches.

However, it is also clear that similar approaches are used in a growing number of systems, which seems to point to a very promising approach. In our work, we will also work on the same direction. The main difference lies in the fact that we intend to explore speculative execution, which should allows us to improve existing designs.

In this comparison, we have not included the system presented in [9], as it uses a lazy replication strategy, thus focusing on different problems and using different techniques. We have also excluded the Gorda system [12], as it is mainly an approach to allow the use of different alternatives solutions for existing database systems.

	Architecture	Replica consistency (client view)	Isolation	Update propagation	Group Communication	Conflict Detection
Kemme et Al. [4]	Distributed	Yes	Various	Both	Yes	System
Ganymed [5]	Primary-copy	Yes	SI	Lazy	No	DB
GSI [6]	N Replicas + Centralized Certifier Or Distributed Certifier	Yes	SI	Eager	No, using the Centralized Certifier. Yes, using the Distributed Certifier	System (Certifier)
Tashkent [7]	N replicas + 1 central certifier	Yes	SI	Eager	No	System (Certifier)
Postgres-R [8]	Distributed	Yes	Serializability	Eager	Yes	System
DSMA [10]	Distributed	Yes	Serializability	Eager	Yes	System (Certifier)
DSMA* [11]	Distributed	Yes	Serializability	Eager	Yes	System (Certifier)

Table 1: Comparing some of the presented systems properties

3 Design

This chapter presents the design of our middleware system for database replication. We start by discussing the setting for which our system is designed and then proceed by presenting the system architecture.

3.1 Analysis

3.1.1 Design Principle

A large number of different applications are built on top of database systems. These applications may have very different transaction workloads, with a different balance between read-only transactions and read-write transactions. For different workloads, different system implementations may behave differently, with some solutions working better for some specific workloads. Our system is no exception to this rule.

We have designed our system to work with any workload given but we have optimized it for working better with workloads with a larger number of read-only transactions than read-write transactions. To this end, we have decided to optimize the execution of read-only transactions. This workload is common in many applications that use databases for storing data, such as web-based applications [5].

As further detailed later, our system replicates the database over a set of replicas using a primary-copy approach, where the primary server is used to execute the read-write operations. Secondary replicas are used to execute read-only transactions. In this approach, with a larger number of read-write transactions, the primary replica becomes a bottleneck and the work done by the secondary replicas is minimal. However, if the number of read-only transactions is larger, the execution of these transactions can be distributed among database replicas, thus balancing the load of the system and allowing a better performance.

This load distribution is only possible by the use of the snapshot isolation semantics that allows read-only transactions to execute in a replica without any synchronization with

other replicas. If full serializability was used, expensive synchronization algorithms would be necessary. The use of the snapshot isolation semantics further improves performance, as read-only transactions are not blocked nor aborted by read-write transactions, therefore further improving the number of read-only transactions that can execute concurrently.

3.1.2 Optimizations

As discussed in [4, 5, 8, 9, 10, 11], a replicated database system that uses a synchronous replication approach may present performance deficiencies when compared to a system with no replication. Besides the basic replication design presented before, we have decided to rely on speculative execution to further improve database performance and reduce the latency for operation execution. Thus, a client may use speculation to avoid the waiting experienced when sending operations to the databases on the remote replicas. By returning the probable result for the operation, the client application can continue executing, thus performing useful work instead of being blocked waiting for the operation result. If the guessed result is correct, this approach can reduce the time necessary to run the client application and improve the performance of the overall system. If the guessed result turns out to be incorrect, the transaction just needs to be aborted.

Since our approach is based on a replicated system with primary-copy using a synchronous approach, the updates applied to the primary replica are propagated to the remaining replicas when the transaction commits. To increase the performance of the system, the result of the commit may be returned to the client before every replica is updated, like in an asynchronous approach. If the primary does not fail, this approach can still provide the illusion of a single copy if the client never observes inconsistencies between the database replicas – this can be achieved by guaranteeing that the client will never start a transaction in a replica that has still not executed this previous updates.

However, in the presence of failures, not updating all the replicas before answering to the client can lead to durability issues. If a primary replica returns the result of a commit and then fails before the updates of the transaction have been applied on another replica,

these updates will not prevail, but the client assumes they had since it has received the answer from the primary replica confirming that the commit succeeded. To avoid these problems, we have defined a minimum of replicas to update after every commit. Thus, the primary replica only replies to the client confirming a commit after this minimum number of replicas have been updated. If the primary replica fails, an updated secondary replica may take its place and the updates prevail. This approach works fine if, at most, the defined minimum number of replicas fails (including the primary).

3.2 Architecture

The architecture of the system is composed by three types of entities: Client, Primary Replica and Secondary Replica. When the system is executing, there exists a single Primary Replica and a limited number of Secondary Replicas. The number of clients is unlimited (although the implementation may impose some limits). Clients run users applications that access the database by communicating with both Primary and Secondary Replicas as explained later.

The Primary Replica maintains the primary copy of the database and executes the read-write transactions. The secondary replicas host copies of the primary database and execute the read-only transactions.

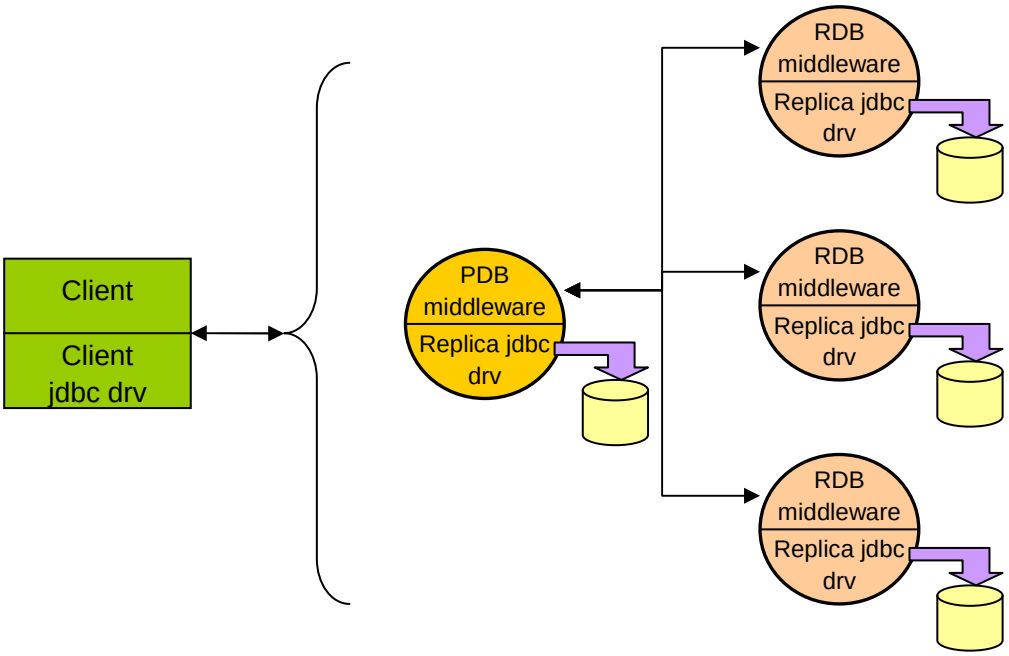

Figure 6: Middleware architecture

The architecture of the system depicted in Figure 6 is an example of the system with just one client that communicates with the middleware, which is composed by a primary server and three secondary servers.

3.2.1 Client

Usually, a Java application that accesses a database system uses a JDBC driver to issue SQL commands using the JDBC defined functions. The JDBC driver contacts directly the database server. In order to allow applications to use our system without modification, we have implemented a custom JDBC driver that acts as a client for our replicated database system.

This approach allows us to use replication along with some other features of the system. The replication is not visible to the client application, and only the client driver is aware of the existence of multiple database replicas.

In our design, read-only transactions are processed by secondary replicas (if any is active) and read-write transactions are processed by the primary replica. To avoid the requirement of pre-defining a transaction as read-only like the Ganymed [5] system, we

use a strategy to emulate the same functionality as that system, where the read-only transactions are executed on the secondary replicas. The first time the client starts a transaction, the client driver asks the primary replica for a secondary replica to process the transaction. A secondary replica is chosen randomly and the client driver uses that replica to execute its read-only transactions. When executing a transaction, if any of the operations is a write, then the transaction continues to be processed by the primary replica instead, as depicted in figure 7. After a transaction is committed the client always begins the execution of the next transaction on the chosen secondary replica. To ensure that the transaction executes in the same version of the database the middleware starts the transaction on both the primary replica and the secondary replica on the beginning of a transaction. If the transaction changes its execution to the other replica it continues executing on the same version of the database.

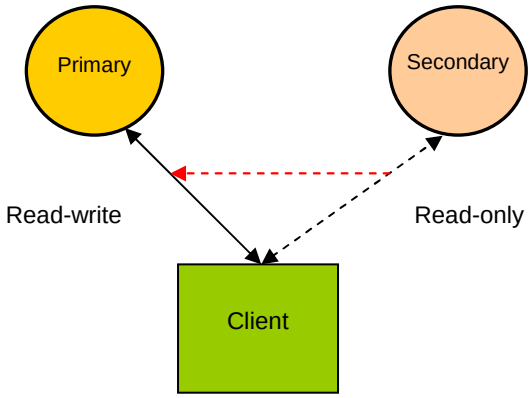

Figure 7: Change of replica when the transaction is found to be read-write

As already mentioned, the system uses speculation to improve its efficiency. The JDBC driver used by the client was developed to be able to communicate with the database middleware using speculation when it is justified. Some operations can be speculative. This means that the client driver sends the operation to the server and before receiving the reply it speculates the most probable result, allowing the application program to continue its execution without waiting for the answer from the server. If latter the reply from the server is different from the one returned to the application, then the transaction needs to be aborted. When the client issues the next operation, or at least when it issues the commit operation, the information about the abort is reported to the application.

Note that this approach of returning an erroneous result to an operation does not present a problem because a transaction can only be considered definite after it executes a commit successfully. Moreover, the transaction is rolled back and the client receives an exception to know that the transaction failed and may re-execute it or not.

3.2.2 Server

The system is a middleware composed of various replicas of the database. There is a primary replica that is used to execute updates to the database, which are sent to the remaining secondary replicas after the local commit on the primary server (replica). As already mentioned there is a defined minimum of replicas that need to be updated before the primary server replies to the client driver. This guarantees that the updates prevail even if the primary server fails (Durability).

The primary replica executes the updates and deals with the transaction conflicts at the database level, i.e., if two concurrent transactions conflict with each other, one will be aborted by the database system. After a transaction commits, the write set of the transaction is propagated to the remaining replicas for updating them. The primary propagates the write sets in commit order, thus guaranteeing that all replicas evolve through and to the same state.

For being able to propagate the write set the updates must be extracted from the database. The updates of a transaction are grouped in a write set by the order they were executed on the database, and a replica that receives the write set executes the updates it contains by the same order to achieve the same database state as the primary replica.

The main role of a secondary replica is to work as a backup of the database, which can be used to perform read-only transactions. Although the system tries to keep all replica synchronized all the time, the secondary replicas can be outdated for some small periods of time. If the transaction is not defined as read-only, the system assumes the transaction is read-only until the client issues a write operation. Therefore, a transaction is first executed on a secondary replica, if present, until the first write operation, and then it continues its execution on the primary replica.

This approach follows the strategy of primary-copy where the writes are only executed on a primary replica. In order to maintain the correctness of the transaction execution, a transaction is both initiated on the primary and in the secondary replica in the same database state, so that it can continue its execution on the primary when a write operation is issued. Otherwise, when changing the execution from the secondary to the primary, different database versions could be used leading to incorrect behaviour. By initiating the transaction in both replicas, we guarantee that the transaction always executes on the same snapshot of the database, even after transferring its execution from the secondary to the primary replica.

To implement this functionality, a straightforward approach is to start the transaction in the primary, and requiring the primary to propagate to the selected secondary replica the information about the transaction start. If the propagation of this information is ordered with the propagation of the write sets from committed transactions, it guarantees that a transaction is started in the same state in both the primary and secondary replicas[1].

Both types of replicas are subject to failures. When the primary server fails any secondary replica can be elected as primary. The secondary replica contacts the other replicas to update itself to the last version of the database if needed. Then it assumes the role of the primary server. The failure of a primary replica is detected by a client or secondary replicas that probe the primary replica and cannot establish contact with it after a defined number of tries. We consider the primary as faulty after some tries to discard sporadic network failures. The component that discovers the faulty primary starts an election algorithm which defines the first active replica as the new primary. This replica contacts all the remaining active replicas to update itself to the last version of the database and only then assumes the role of the primary replica.

[1] When using JDBC, there is no explicit start transaction operation – a transaction is started when the first operation that requires database access is executed. We force the start of a transaction by reading some data element, thus using a slightly older than necessary snapshot for transaction execution.

When a secondary replica fails, it is discarded as an active replica by the primary server, which then warns the remaining replicas of their *sibling* failure, for it to be removed from their lists. The failing replica can be offline for a small or large period of time, or simply to be offline forever. When, and if, it returns online, it must be updated before it can be used to communicate with the clients. For that matter, it asks a secondary replica or the primary replica if no other replicas are present, for the latest unseen updates. The first time a secondary replica becomes online, it asks a secondary replica, or the primary when the primary is the only replica online, for the whole database, creating a local copy of the database.

Both the primary replica and the secondary replicas log the write sets of each version of the database in main memory, to send them to a recovering replica which needs to update its database to the current global version. We opted to keep the logs in main memory instead of writing the logs to secondary memory to avoid decreasing the performance of the system. Our solution keeps the logs in main memory, but they should be garbage-collected after a while. This brings another issue illustrated by the following example. One of the secondary replicas becomes offline and is considered faulty. The remaining replicas are then updated and the system discards later discards the updates. If the failed replica returns online must be updated to the same database version as the other replicas, but the missed updates are no longer logged. A practical but inefficient solution is to propagate the whole database for the replica to be updated.

3.2.3 Speculation

Speculation is a technique used to improve efficiency and is based on executing with probable values instead of certain values, that is, values which may later be proven to be incorrect are used before knowing the correct values. We have applied the use of speculation on the client for hiding the latency of communication with the servers. The client application does not know about the existence of speculative operations, only the driver used by the client to communicate with the middleware sees the speculation.

Some operations are eligible as speculative since their result can be predicted. Assuming the most probable result, the client driver returns the answer immediately to

the client application even before contacting the server. The operations elected as speculative were the creation of a prepared statement, the close of a connection, the execution of a statement, and the execution of a prepared statement. Each of these operations was chosen since its results can be predicted. Therefore the client does not need to wait for the reply from the server to continue with the execution.

Regarding the prepared statements there is also caching of the values to be set on the statements that are sent to the server only on the execute operation, instead of communicating with the server for each value to be set. This avoids some remote calls, which require some precious time.

4 Implementation

In this chapter, we present the most important implementation details of our prototype. We have organized this chapter in the following sections. Section 4.1 exposes the base functionality of the system, and is sub-divided in five chapters. Chapter 4.1.1 explains what happens when the system is initialized. Chapter 4.1.2 explains how the transactions are executed and processed by the system and all the consequent operations. Chapter 4.1.3 exposes how the system handles the JDBC structures. Chapter 4.1.4 deals with the concurrency and the use of speculation and Chapter 4.1.5 explains how the replicas remained updated while the system is executing. Section 4.2 discusses the extraction of the write sets and Section 4.3 shows how the system deals with failures. Finally Section 4.4 explains how the system uses and implements speculation.

4.1 Base Functionality

The system was implemented in Java and is prepared to be used with the PostgreSQL database system. But with some minor changes it can be used with any database system that provides Snapshot Isolation and which can be accessed through JDBC. These minor changes are related with the creation of triggers, which is not standard SQL and is implemented differently for each database system.

4.1.1 Binding

The replicas implement the Remote interface so they can be registered on the RMI registry to accept RMI remote calls. A primary replica registers itself on a RMI Registry and becomes receptive to receive requests from the clients. A secondary replica also registers itself on a RMI Registry and tries to contact the primary replica based on an initial defined list of possible hosts that may host the primary replica. This list of

possible hosts includes the hosts that may have a server and is used for probing of the primary replica by the client driver or a secondary replica, as well as probing other secondary replicas. This list is used to initialize a list of active replicas that is present in every replica. The list of active replicas is dynamically updated when a new replica arrives or an active replica becomes inactive. When it starts, a secondary replica registers itself on the primary replica so that it can be updated and be used to reply to clients requests.

As mentioned in the design section, the client of the system is actually the JDBC driver used by an application to access the database. The users' applications just issue SQL commands through the JDBC driver. On request of the client application, the JDBC driver opens remote connections to the database middleware, which then uses to issue operations to the database. The driver uses the list of possible replica hosts to find the primary replica to start executing.

4.1.2 Transaction Execution

As it was explained in the design of the middleware, when a client starts a transaction, it starts executing operations on a randomly chosen secondary replica. But the execution of the transaction may change to the primary replica if the transaction updates the database. For allowing this change during transaction execution, when a client starts a transaction, it must start a transaction on the both the primary and secondary replicas. So the transaction is started on both replicas at the same time, using the same database snapshot.

For distributing the load among secondary replicas, the primary replicas simply selects randomly a secondary replica for being used by a client. Thus, when the client creates a connection to the primary, it is the primary that creates a connection to an updated replica and sends it to the client.

To guarantee that a transaction is started in the same version of the database, we start the transaction on both the primary replica and the secondary replica on the beginning of a transaction by forcing the start of a transaction reading some data element. This is

needed because there is no explicit start of a transaction in JDBC. If the transaction changes its execution to the other replica it continues executing on the same version of the database.

When the client starts executing a transaction, the transaction is first assumed to be read-only. Therefore, the secondary replica connection is defined as the default connection and all operations issued by the client use this connection to execute operation just in the secondary replica database. Before sending a new operation to the secondary replica, the client checks if the operation can possibly be an update to the database. Besides common update operations (insert, delete, update), there is a particular select SQL statement that is also an update. This statement is the select for update, which is considered as an update by the system since it blocks the selected rows until commit, and it is usually used to later update those rows. Even if the updates are never executed, blocking rows in the secondary replicas could lead to conflicts when applying the write sets from other transactions, since the rows could still be blocked.

If one of the operations is an update, then the replica being used changes from the secondary replica to the primary replica, and the following operations issued by the client are sent to this replica through the connection that was created initially to the primary. After a commit or rollback issued by the client, the default replica is again reset to the secondary replica.

Before each of the instructions issued by the client is sent to the server, there is a check for the need to abort the transaction. This is needed because of the use of speculation and it will be explain in the respective chapter.

When the client issues a commit operation, the primary starts by committing the transaction locally. After that, it propagates the updates of a transaction as a write set, to the secondary replicas that immediately acknowledge the reception before applying the write set. When the primary knows that a pre-determined number of replicas have received the updates, it will return the reply to the client, as depicted in figure 8. This approach guarantees that the updates will not be lost if at-most, the pre-determined number of replicas fail.

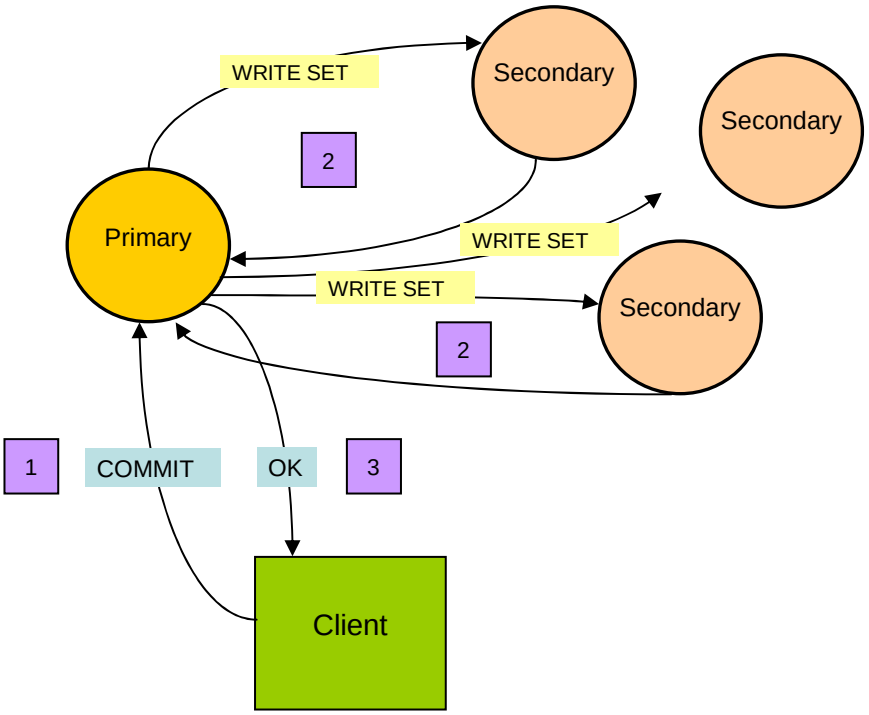

Figure 8: Execution of a commit operation.

When propagating the update to the replicas, it is necessary to ensure that the updates will be executed in the secondary replicas in the same order as in the primary. For that we have created a table on the database to register the version of the database (the version number is also used for failure recovery as explained later). Each new committed transaction increases the version. When a write set is propagated to the secondary replicas it is stamped with the corresponding version. Then, the secondary replicas apply the write sets by the increasing order of version, as depicted in figure 9.

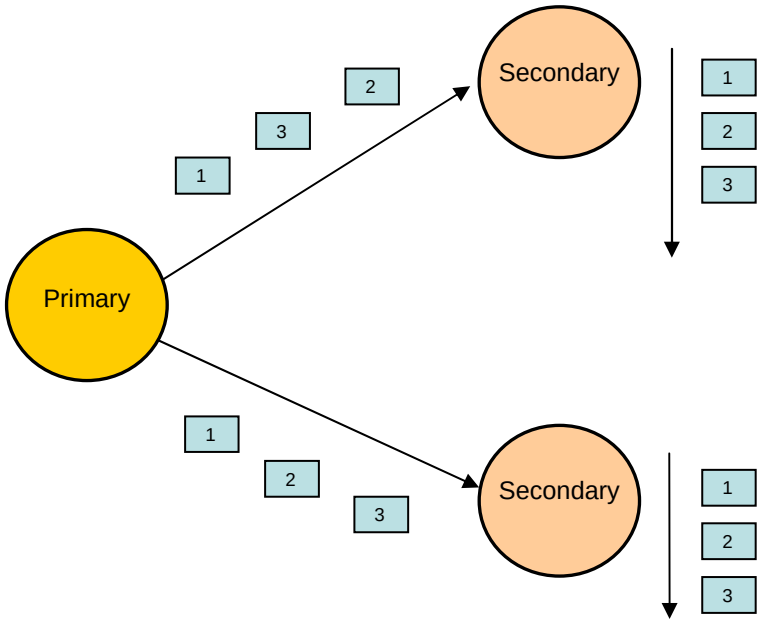

Figure 9: Reordering of the write sets on the secondary replicas

4.1.3 Handling JDBC Internal Structures

The JDBC driver uses a set of internal structures which do not imply communication with the database system. By using a middleware system the propagation of these structures to the client or to the server would result in a remote call. Therefore we use a technique which avoids the propagation of these structures. The structures are Statements, PreparedStatements and ResultSets.. The client driver creates proxies to these structures kept on the servers. The client issues operations on these structures, so the structures must be identifiable on the server. The proxy of the structure keeps an identifier which is sent to the server for each operation on this structure. For the ResultSet structure, the server creates an identifier and replies it to the client upon the creation of one structure of this type. For the Statements and PreparedStatements, it is the client driver itself that creates the identifier and sends it to the server, because these structures need to be created both on the secondary replicas and on the primary. This approach allows the immediate use of the Statement and PreparedStatement structures in the client, without having to wait for the creation of the structure on the server. As these operations (almost) never fail, this approach allows avoiding some performance penalty. If they fail, the transaction is later aborted.

This is mostly useful for PreparedStatements, since all its operations except the executeUpdate, executeBatch and executeQuery, are executed locally on the client driver, as it will be explained next, and can start to execute without the need of contacting the server.

All operations on any of these structures are propagated sending the identifier and the additional data needed to perform the operation. Obviously, this is hidden from the client application to maintain the transparency of the remote operations.

Regarding the need to create the Statement and PreparedStatement structures on both the primary replica and the secondary replica, the reason for this is precisely because the execution of the transaction may start on the secondary replica and then be transferred to the primary replica and after commit or rollback return to the secondary replica. So, there may be operations issued over the statement when the execution is on the secondary (resp. primary) that need to be also reflected in the primary (resp. secondary) for the following transactions.

The PreparedStatements and the ResultSets do some caching of data. A PreparedStatement is a precompiled SQL statement for efficient execution multiple times. It provides methods to change the values of the statement after its creation. With a remote server, the remote calls are a burden, so they must be avoid. So, the setter methods are registered on the client side and only propagated when the execution of the statement is required, in a single remote call to the server.

A ResultSet is a table of data representing a database result set returned from the execution of a statement that queries the database. Like the PreparedStatement, a straightforward approach would be for each operation on the ResultSet to contact the remote server. Once again, we avoid unnecessary remote calls by bringing the table of data of the ResultSet to the client, so that it can be used to execute operations of the ResultSet locally. A future improvement could be to balance the number of rows brought instead of bringing the entire table to the client.

4.1.4 Handling with Concurrency and Speculation Consequences

When sending the requests to the servers, it is possible that the order which was issued by the client application is different than the order the requests arrive at the server because of the speculation. The reason why this happens will be explain further on the section dedicated to speculation. So the server must implement a strategy for guaranteeing that the operations from a given transaction are executed in the correct order. The implemented solution is quite simple. Since the reordering is only needed for the instructions issued on the same connection, we stamp the requests with a serial number, and the server executes the instructions by the correct order. To achieve this requirement, the instructions that try to execute before time are blocked till their turn arrives.

4.1.5 Keeping the replicas updated

A secondary replica must register itself with the primary when it becomes online. If it is the first time the replica enters the system the whole database must be copied to the replica. It is preferable to use a secondary replica for this procedure to lower the load on the primary replica. This copy can be a very heavy procedure if the database is very large. With that in mind the strategy adopted was to bring to primary memory the ResultSet with all the rows from a table, for each table at a time, and send the rows individually to the new secondary replica. Since a table can be quite large the rows are brought to primary memory by chunks. When a chunk has no more rows it is filled up with more rows, until all rows are propagated to the secondary replica. This is done for every table from the database, except the table that retains the version of the database, which is a table specific from the system.

Each server has a database version number saved on a specific table. The version is increased with each commit to the database. The use of a version number is useful for various situations, for example when a secondary replica fails and later becomes online again it must be updated with the missed updates. This is done based on the version number of the failed replica, since only the unseen updates between the version number

of the failed replica and the global system database version must be propagated to the replica.

The primary sends the write set to all active replicas and waits for a defined minimum number of replicas to be updated before returning the result of the commit of the transaction to the client.

4.2 Write-set extraction

As it was already mentioned, the updates to the primary replica are propagated to the secondary replicas as write sets. For the write sets to be propagated they must be extracted from the database. During the study of the extraction of write sets we have made an overview of the possibilities and after some testing chose the most efficient solution available.

We found three approaches for extracting the write sets. The first one was to use a database with implicit write set extraction. This is not supported in the database we wanted to use (Postgresql), therefore this solution was immediately rejected.

The second solution would be to register the update SQL instructions issued by the client and forward them to be executed in the remaining replicas. This solution presents a problem since some instructions might be non-deterministic, mainly when they trigger triggers that update the database. For example, instructions that require the use of the replica's current system time will return different values. If these instructions are executed on different replicas, the results in one replica will be different from the ones obtained on another replica, leading the database to become inconsistent.

So we have opted for the last approach for the extracting write set: recording the actual changed performed in the database. To this end, we rely on the use of triggers.

We have created a script that reads the database and for each table with primary key creates a trigger for updates, deletes and inserts on that table. This means that all the database tables must have primary keys, which is a common requirement when using database replication and can be easily addressed in tables that do not include such

feature by internally adding an additional column. For example, we have to do this in the table History of the benchmark TPC-C.

Each of these triggers registers all the updates to a specific table. There are two possible approaches to send this updates to the system. One is to register the updates on an extra table and on commit (when the write set is required to be sent to the replicas) the data from this table is extracted into a write set, and sent to the replicas. The other approach is to use the RAISE SQL function to propagate the updates to the system immediately when they are executed. The primary replica stores the warning thrown by the RAISE function on the write set. On commit, the write set is simply sent to the replicas since it has all the updates executed on the transaction. Both strategies were implemented and tested, and the results have shown the latter strategy (the RAISE strategy) to be the most efficient, and consequently the one chosen for the implemented system.

4.3 Treatment of Failures

When a replica tries to contact another after a defined number of tries without success that replica is considered to be faulty. The system takes into account the failures of the replicas, either the primary or the secondary.

When a secondary replica fails, it may be inactive for a very long time or even to never be back online again. So a failed replica must be removed from the active replicas list that exists in the primary replica and remaining active replicas. When a failed secondary replica is detected, usually when the primary tries to contact it, it is removed from the primary list of active replicas, and then the primary contacts the remaining active secondary replicas for them to remove the failed replica from their respective lists of active replicas.

When a failed replica returns online, it must be updated to the current version of the database before it can be used in the system. To this end, when a secondary replica returns, it registers itself with the primary server and this one chooses a secondary replica for it to propagate the unseen updates – we call this replica, the sending replica.

If there is no other replica, the primary itself propagates the updates. While on this procedure the replicas cannot be used by clients.

For making this procedure as efficient as possible, the returning secondary replica sends its last version to the sending replica. The sending replica only sends the unseen updates after that version. The receiver replica then applies the updates in the correct order and finally registers itself as an active replica with the primary replica. The primary replica adds the replica to its list of active replicas, saving also its current version, and contacts the other replicas to add the returned replica to their lists of active replicas. In the meanwhile the returning secondary replica buffers the new write sets sent by the primary while it was being updated, and after executing the missed updates applies the new write sets.

If a primary replica fails then the procedure is different. The failure of a primary is detected by a client that tries to contact the primary or by a secondary replica that tries to do the same. If after a defined number of tries the primary replica continues unreachable, then it is assumed to be offline and an election algorithm is executed. This algorithm is called by the entity that discovered the failure in the primary replica. The first secondary on the list of active replicas tries to elect itself by majority vote. It contacts the other active replicas warning them of the candidacy and after receiving votes from a defined number of replicas (must be equal or greater than half of the active replicas), it assumes the role of primary replica, as depicted in figure 10.

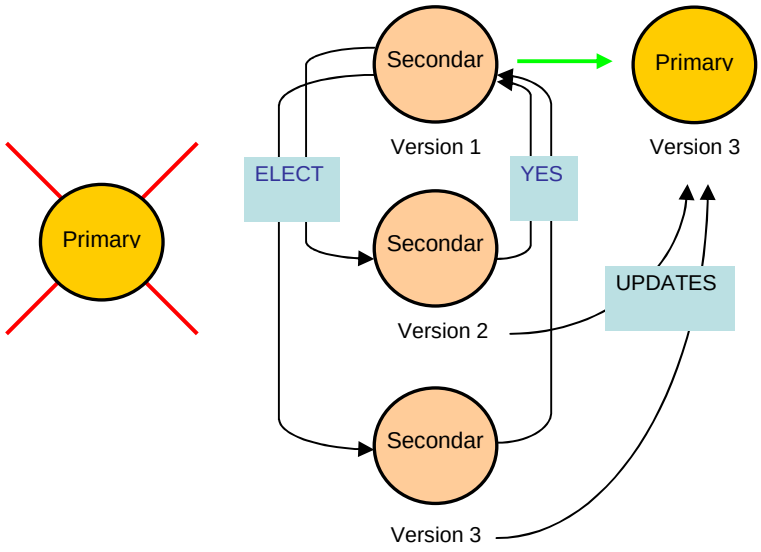

Figure 10: Election of a secondary replica to substitute failed primary

During the election, each secondary replica sends the updates not seen by the candidate replica, for it to update itself to the latest version of the database before assuming the primary role.

4.4 Speculation

As it was already said, the result of some operations can be speculated in the client, so that it can continue its execution without waiting for the server reply, thus possibly improving performance. The operations that were chosen to be speculative, because their results could be predicted, were the following.

- The close of a connection can be speculative since no other operation over this connection will be issued and there is no need for it to close on the database to continue the execution.
- The creation of a PreparedStatement which we assume to succeed, allowing its immediate use on the client side to start executing operations on the prepared statement, since most of these operations do not need to contact the database.

- The execution of a statement can be an update or a query. If it is an update the result is the number of the execution is a Boolean which states if the statement is a query or not and the SQL instruction can be processed to guess which will be the reply to the client. If it is immediately executed as a query then a result set object is returned. Since this object may not be immediately used we create can return an empty object and only fill it when the answer from the server is returned.

- Finally the execution of a prepared statement can also be speculative. If it is a query the operation is the same as for a regular statement. If it is an update it is returned the number of rows updated which we assume to be 1. If the prepared statement is executed as a batch of commands then an array of updated row counts is returned with the value returned for each command, which can also be predicted.

When one of these operations is issued by the client the operation is added to a list of speculative operations to be executed and its probable result is returned immediately to the client. These operations are then executed in background and removed from the list.

The use of speculation presents a problem with the ordering of the instructions. Since a speculative operation is executed in background and the main execution on the client continues, it is possible that a posterior operation begins to execute at the server before this speculative execution which was previous issued by the client application. Therefore the operations must be stamped with a serial number. On the server they are reordered by blocking the operations that try to execute before their time until their time to execute arrives. The client must have two different serial numbers for each of the servers, the primary replica and the secondary replica being used. These serial numbers are created and stored when a connection is created. Then for each operation issued to a server the corresponding serial number is sent and incremented. Therefore each server has its own serial number also.

There is a subtle, problem presented by the use of speculation. When a speculative operation returns its result to the client, the execution of the operation on the server was

not executed yet. If the execution on the server fails or it has a different result, then the transaction must be aborted. The problem is that it cannot be aborted in the speculative operation since it already returned the assumed probable result. So, the solution is to use an abort flag. If some problem is detected related with the speculative execution, the transaction is immediately rolled back in the servers. The client driver sets the abort flag. Before each client operation is propagated to the server, the flag is checked. If the flag is set to abort, then the client driver launches a `SQLTransactionRollbackException` to the client application, notifying the client that some problem has occurred during transaction execution.

In this case, there still might be client operations from the aborted transaction in transit on their way to the server. To avoid executing these operations the server and the client driver have a counter that both increase after an abort of a transaction and the client sends the value of its counter when sending an operation to the server. When the messages of the aborted transaction arrive at the server they have a value different from the value on the server, since it was already increased. These messages are simply ignored and only the messages with the current value are executed.

5 Results

5.1 Write-set extraction

As mentioned on the implementation chapter, we have tested two ways of extracting a transaction write set. The first consisted in using a special table to register all the updates using a database trigger. On commit, the values stored on the table were returned to the application. The second method consisted in using a trigger that raised a warning to the JDBC driver with the update itself, each time an update (insert, update or delete) is performed.

	No extraction	Extraction with table	Extraction with Raise
Write-ahead log	1287.2 (ms)	1979.8 (ms)	1519.6 (ms)
No write-ahead log	1021.4 (ms)	1830.6 (ms)	1371.8(ms)

Table 2: Time measures (milliseconds) for the extraction of the write sets

To evaluate both methods, we have run a small micro-benchmark, with transactions consisting in an insert, a select and an update operation. Table 5 presents the time measured for executing one hundred transactions, dropping and creating the used table before executing each transaction. Each test was executed five times and the results presented in Table 5 are the average between the five executions.

The tests were performed with the write-ahead log active and inactive. The write-ahead log is a mechanism used by the database system to provide atomicity and durability. Changes to the data files are logged to permanent storage before being executed so that

in the event of a crash the database can be recovered using the log. Without the use of a write-ahead log the data pages do not need to be flushed to disk on every transaction commit, thus not ensuring the durability property.

We observe that the overhead of extracting the write set using the Raise warning approach is minimal (almost the same time as the one obtained without extraction), while the overhead of using a special table is considerable. Therefore the Raise warning approach was chosen.

5.2 Benchmark TPC-C

Our system was benchmarked with the on-line transaction processing (OLTP) benchmark TPC-C. The benchmark simulates an environment where a population of terminal operators executes transactions on a database of nine tables. The benchmark is composed by five OLTP transactions that use primary and secondary key accesses. The five types of transaction defined are:

- New Order – enters a new order from a customer;
- Payment – updates the customer balance to reflect a payment;
- Delivery – delivers orders (batch transaction);
- Order-status – retrieves the status of the most recent order from a customer. This is a read-only transaction;
- Stock-level – monitors a warehouse inventory. This is a complex read-only transaction.

The system was only evaluated in terms of the benefits of using replication. The speculation benefits were not observed since the TPC-C Benchmark just executes database operations with no processing between these operations, eliminating the greatest strength of this type of speculation. The speculation is good to advance the

local execution while the remote calls are being processed, avoiding the waste of processor cycles while waiting for the answer to the remote call. Since there is almost no local execution in the benchmark this advantages of this execution are not reflected on the results obtained. The time schedule did not allow to experiment with a different Benchmark since that would imply a rewriting of the client code to remove all the speculation to be able to compare the execution using speculation and without speculation.

The benchmark default transaction distribution is: 45% New Order, 43% Payment, 4% Order-status, 4% Delivery, 4% Stock-level. This means 92% of the transactions are read-write while only 8% of them are read-only transactions. This is clearly not a good setting for our system. So we have opted to use the following distribution: 4% New Order, 4% Payment, 45% Order-status, 2% Delivery, 45% Stock-level.

The middleware was benchmarked with a variable number of clients: three, four (maximum of servers on the system) and eight (double the maximum of servers), and a variable number of servers from one to four (the servers are a primary replica and secondary replicas). The benchmark was executed for a period of 20 seconds for each different configuration (one client and one server, two clients and one server, etc.)

We executed the benchmark with the PostgreSQL Serializable isolation level, which corresponds roughly to the Snapshot Isolation level.

		Servers			
		1	2	3	4
Clients	3	2881,667	+307,6667	+341,3333	+570
	4	3285,4	+500,9333	+751,2667	+928,9333
	8	3612	+1082,333	+1783,667	1904
	3		+11%	+12%	+20%
	4		+15%	+23%	+28%
	8		+30%	+49%	+53%

Table 3: Total of executed transactions with the increase of the number of servers

Table 3 shows that the total number of executed transactions increases with the increase in the number of servers for the same number of clients. Since the workload is composed mainly by read-only transactions, and this type of transactions is executed by the secondary replicas (while the read-write are executed by the primary replica), the execution of the read-only transactions is decentralized. By decentralizing de execution of the read-only transactions a greater number of read-write transactions can be executed by the primary replica and also a higher number of read-only transactions is executed. The latter is increased by increasing the number of secondary replicas.

6 Conclusion

6.1 Critical evaluation

Database systems are a key element in a large number of different applications. Thus, it is important for database systems to be reliable and scalable. To achieve these properties, replication is an important feature.

In this work, we have implemented a middleware system for database replication following a synchronous approach to avoid replica divergence. From the point of view of a client, the system provides single-copy consistency since the client executes as if only one copy of the database exists. Internally, this is not the case since when the result of a commit is returned to a client there are replicas that might have not been updated yet.

Clients use a custom JDBC driver, which we have implemented, to contact with the middleware, thus allowing unmodified applications to use our system unmodified. The middleware is composed of a primary replica and several secondary replicas. Replicas use the database JDBC driver to communicate with the local database system (PostgreSQL in our implementation).

The database is fully replicated in all the replicas. Execute replica executes on snapshot isolation, the isolation level that we find to better apply to our system. We think it increases the efficiency of the execution of read-only transactions. Besides it avoids the same phenomena as the serializable isolation level.

The system is based on a primary-copy architecture where the primary-copy only executes update transactions while the secondary replicas execute all the read-only transactions. The system architecture is similar to the Ganymed [5] system, with some

modifications. First, we avoid the need of declaring the read-only transactions as read-only beforehand, still executing them in a secondary replica. Second, we implement a speculative mechanism on the client to try to improve system's performance.

The use of primary-copy is usually associated with two disadvantages. The primary-copy is a bottleneck and a single point of failure. The bottleneck problem is negligible when the fraction of read-only transactions is substantially higher than that of read-write transactions. To avoid the primary copy to be a single point of failure, we have implemented a failure detection strategy with an election algorithm to elect an updated secondary replica as primary replica. The system also processes the failures of the secondary replicas. When a failed secondary replica returns online, it uses another replica (preferably a secondary replica) to get the missed updates before becoming an active replica that can be used by the clients.

Another contribution of this work is related with the extraction of write sets. We tested two alternatives and presented the results to justify the choice taken. We have also implemented an automatic script to read the database tables and create the triggers related with the extraction.

The system also uses speculative execution for some of the client remote calls to improve the efficiency of its execution. The speculation could also theoretically be used on the servers but to implement it we needed extra time.

We intended the middleware to be efficient. The testing of the middleware itself was not conclusive enough to prove this was achieved. The system was benchmarked with a workload with a much larger number of read-write transactions than read-only. This does not benefit the system implemented efficiency since it was implemented for workloads with a much larger number of read-only transactions. The system needs a more exhaustive test, with the use of a workload with a greater load of read-only transactions than read-write transactions. Also by testing the impact of using speculation some additional conclusions may be drawn. This last test may be executed testing the system with a different benchmark that includes other processing besides the database operations.

Nevertheless the results obtained show one benefit of using replication. The percentage of committed transactions increases because the burden over the primary replica decreases. And the benefits of using replication still prevail. If a server fails there are other servers ready to process the client's requests. A secondary replica can assume the role of a failed primary replica and the system is maintained online. So the replication improves the availability of the system.

6.2 Future work

The implemented system has some limitations that can be improved in the future. But first it has to go through more testing.

The benefits of using speculation must be evaluated using a benchmark which has processing other than database operations. The system itself should also be benchmarked with different benchmarks like the TPC-W to better evaluate its capabilities and efficiency.

The system can be improved in several aspects in order to optimize it. We present some ideas to implement in the future.

As previously mentioned we could balance the number of rows brought from the server when creating a ResultSet instead of bringing the entire table.

The replica failure recovering could be improved. The election algorithm, executed when the primary fails, chooses the first active replica as the new replica, but it could chose the replica with the most recent updates. Also, when a secondary replica fails and returns online it must be updated to the last version of the database. The log approach could be more efficient in terms of memory space since all logs are being kept on main memory and no garbage collection is being done. We could implement a garbage collection that would clean the logs from all replicas after a defined time limit was achieved and only if all the active replicas had the version of their databases with a number equal or higher than the log entry to be deleted. If a failed replica returned

online it would update itself from the logs if the unseen updates were still logged, or else would request the whole database from a secondary replica.

An interesting improvement to implement is related with the speculation. When a client uses speculation it issues an operation to the server continuing the execution and can immediately issue another operation to the server. On the server side the first operation starts to execute and the next operation must wait its turn to execute and only then is executed. If a third operation is issued it must also wait for its turn to execute, and so on. So the server sends the operations to the database one by one. But instead we can group the operations that arrive at the server and are waiting for their turn to execute and send them as a batch of operations to the database. In other words, the operations that arrive while one prior operation is being executed are batched to be sent to the database, instead on sending them one at a time.

Regarding speculation there are some other improvements that may be studied, like the use of speculation on the servers to communicate with the client or between the servers. Other possible improvement is to apply the speculation to the commit operations, on the client. But this presents a problem. If we speculate the commit of a transaction, it can no longer abort since the results of the transaction can no longer be repealed. We need to use a system like the Speculator [1] or one that presents the same functionality.

Finally we could investigate different ways of distributing the workload of read-only transactions by the secondary replicas instead of just choosing a random replica.

All the above improvement proposals should be accompanied by the respective tests.

7 Bibliography

[1] Edmund B. Nightingale, Peter M. Chen, Jason Flinn: *Speculative execution in a distributed file system*. In Proceedings of the twentieth ACM symposium on Operating systems principles, 2005: 191-205, 2005.

[2] André Abecasis G. Ferreira. *Execução Especulativa em Bases de Dados*. Departamento de Informática, FCT-Universidade Nova de Lisboa, 2007.

[3] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, Edmund L. Wong: *Zyzzyva: speculative byzantine fault tolerance*. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 2007: 45-58.

[4] Bettina Kemme, Gustavo Alonso: *A Suite of Database Replication Protocols based on Group Communication Primitives*. In Proceedings of the The 18th International Conference on Distributed Computing Systems, 1998: 156-163.

[5] Christian Plattner, Gustavo Alonso: *Ganymed: Scalable Replication for Transactional Web Applications*. In Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, 2004: 155-174.

[6] Sameh Elnikety, Willy Zwaenepoel, Fernando Pedone: *Database Replication Using Generalized Snapshot Isolation*. In Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), 2005: 73-84

[7] Sameh Elnikety, Steven G. Dropsho, Willy Zwaenepoel: *Tashkent+: memory-aware load balancing and update filtering in replicated databases*. In Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems, 2007: 399-412.

[8] Bettina Kemme, Gustavo Alonso: *Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication*. In Proceedings of the 26th International Conference on Very Large Data Bases, 2000: 134-143.

[9] Jim Gray, Pat Helland, Patrick E. O'Neil, Dennis Shasha: *The Dangers of Replication and a Solution*. SIGMOD Conference 1996: 173-182.

[10] Fernando Pedone, Rachid Guerraoui, André Schiper: *The Database State Machine Approach*. Distributed and Parallel Databases 14(1): 71-98 (2003).

[11] Vaid Zuikeviciute, Fernando Pedone. *Revisiting the Database State Machine Approach.* VLDB Workshop on Design, Implementation, and Deployment of Database Replication, 2005.

[12] Afrânio Correia Jr., José Pereira, Luís Rodrigues, Nuno Carvalho, Ricardo Vilaça, Rui Carlos Oliveira, Susana Guedes: *GORDA: An Open Architecture for Database Replication*. In 6th IEEE International Symposium on Network Computing and Applications 2007: 287-290.

[13] ANSI SQL-92. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt