On the Study of Different Approaches to Database Replication in the Cloud

Melissa Santana  José Enrique Armendáriz-Íñigo

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
Campus de Vera, s/n; 46022 Valencia; Spain

armendariz@iti.upv.es

4.6 IMPLEMENTATION DETAILS ................................................................................................................................. 33
5.1 EXPERIMENTAL SETTINGS........................................................................................................................................... 39
5.2 RESULTS ........................................................................................................................................................................... 41
  5.2.1 Response Time results .................................................................................................................................................. 41
  5.2.2 Abortion rate results .................................................................................................................................................. 45
  5.2.3 System throughput results ........................................................................................................................................ 46
6 CONCLUSIONS AND FUTURE WORK ............................................................................................................................... 48
  6.1 CONCLUSIONS ................................................................................................................................................................. 48
  6.2 FUTURE WORK ................................................................................................................................................................. 48
List of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Caption</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Replication in database systems</td>
</tr>
<tr>
<td>2.2</td>
<td>Active replication scheme</td>
</tr>
<tr>
<td>2.3</td>
<td>Pseudocode for active replication. Delegate server (left) and any server (right)</td>
</tr>
<tr>
<td>2.4</td>
<td>Certification-based replication scheme</td>
</tr>
<tr>
<td>2.5</td>
<td>Pseudocode for certification-based replication. Delegate server (left) and any server (right)</td>
</tr>
<tr>
<td>2.6</td>
<td>Weak voting replication scheme</td>
</tr>
<tr>
<td>2.7</td>
<td>Pseudocode for weak voting-based replication. Delegate server (left) and any server (middle and right)</td>
</tr>
<tr>
<td>3.1</td>
<td>Process of update propagation in a data partition</td>
</tr>
<tr>
<td>3.2</td>
<td>Proposed structure for the servers</td>
</tr>
<tr>
<td>3.3</td>
<td>Transaction execution according with the freshness level</td>
</tr>
<tr>
<td>3.4</td>
<td>Resources use for the proposed system</td>
</tr>
<tr>
<td>3.5</td>
<td>System model</td>
</tr>
<tr>
<td>4.1</td>
<td>Client pseudo-code</td>
</tr>
<tr>
<td>4.2</td>
<td>Communication module pseudo-code</td>
</tr>
<tr>
<td>4.3</td>
<td>Pseudo-code for lock manager</td>
</tr>
<tr>
<td>4.4</td>
<td>Pseudo-code for database replication module</td>
</tr>
<tr>
<td>4.5</td>
<td>Pseudo-code for server</td>
</tr>
<tr>
<td>4.6</td>
<td>Pseudo-code for metadata manager</td>
</tr>
<tr>
<td>4.7</td>
<td>Class diagram of the proposed system</td>
</tr>
<tr>
<td>4.8</td>
<td>Object interaction for lazy protocols</td>
</tr>
<tr>
<td>4.9</td>
<td>Object interaction for eager protocols</td>
</tr>
<tr>
<td>4.10</td>
<td>Object interaction for active replication</td>
</tr>
<tr>
<td>4.11</td>
<td>Object interaction for certification</td>
</tr>
<tr>
<td>4.12</td>
<td>Object interaction for weak voting</td>
</tr>
<tr>
<td>5.1</td>
<td>Response time depending on the write factor parameter. With no hierarchy.</td>
</tr>
<tr>
<td>5.2</td>
<td>Response time depending on the write factor and freshness parameters. Having hierarchy with write factor=0.75</td>
</tr>
<tr>
<td>5.3</td>
<td>Response time depending on the write factor and freshness parameters. Having hierarchy with write factor=0.5</td>
</tr>
<tr>
<td>5.4</td>
<td>Response time depending on the write factor and freshness parameters. Having hierarchy with write factor=0.25</td>
</tr>
<tr>
<td>5.5</td>
<td>Abortion rate for a no hierarchical architecture.</td>
</tr>
<tr>
<td>5.6</td>
<td>System’s throughput</td>
</tr>
</tbody>
</table>
List of Tables

TABLE 5-1 SYSTEM CONFIGURATION .................................................................................................................. 40
Abstract

Cloud computing is becoming one of the most used paradigms to deploy highly available and scalable systems. These systems usually demand the management of huge amounts of data, which cannot be solved with traditional nor replicated database systems as we know them. However, it is well-known that traditional replication protocols do not scale well for a cloud environment. Moreover, recent approaches propose to restrict the scope of the transactional support in the cloud in order to ensure high availability.

This work presents a study of the traditional replication techniques for static systems applied in a cloud scheme to achieve transactional support combining the high availability and scalability that characterizes cloud systems. This system also maintains the appreciated features of relational databases. This proposal offers different consistency levels according to the demands of client applications using a replication strategy based on a combination of traditional database replication techniques with asynchronous epidemics updates. We have run several simulations that show this is a very interesting approach to provide transactional support to clients with different consistency guaranties while leveraging the resources used. A considerable number of simulations were executed to evaluate the behavior of the replication techniques in the cloud environment based on the response time, the abortion rate and the throughput of the system.
Chapter 1

Introduction

Replicated databases have become very attractive due to an increasing demand for storage systems which provide high scalability and availability. High scalability implies that the database system can grow in the amount of data stored with no penalty in terms of performance, which is measured in the response time to clients’ requests; no matter how many users can be accessing to the same data at the same time. High availability implies that the system will be able to response the different received requests for a very high percentage of the time even in the presence of failures. This is easier to achieve in a replicated environment because data is not kept at a single site, so in case of a failure, the same data can be accessed from another replica.

However, replication presents the problem of keeping all data stored in all replicas consistent. There are several replication protocols to minimize this consistency problem which propose different ways to maintain the data updated in all replicas using some communication tools to propagate the changes. Thus, to reduce the cost of propagating updates to all replicas, two different strategies have been proposed according to its synchronization [2]: eager replication and lazy replication. Eager replication performs the synchronization within transaction boundaries; while lazy replication propagates the changes after the client receives the transaction commit. Eager protocols reach consistency requirement as part of their execution but introduce a higher latency, on the other hand, lazy protocols lead to inconsistencies, which have to be solved later.

With regard to where updates are carried out we distinguish between primary copy [14, 26] and update everywhere [5, 25]. In the primary copy all update transactions are executed in the same replica, the delegate. In this case the synchronization is performed in a straightforward way, propagating the delegate updates to the secondary replicas. For update everywhere the update transactions can be executed in any replica; this implementation presents some issues like conflicts between transactions belonging to different replicas.

Group communication systems have provided a very useful abstraction to be employed when designing update everywhere replication protocols [16, 19, 20]. There are several replications techniques based on one of the primitives of group communication: total order broadcast [2]. The first technique is active replication, which executes the update in the delegate server that issues the total order broadcast on behalf of the client. The second technique is certification-based replication, is optimistic (i.e, operations contained inside a transaction are executed with no restrictions and checks for consistency violations at its end). In this technique all servers execute a certification phase where is decided if the transaction can commit or must abort. And the third technique is weak voting replication, which is similar to certification-based replication, the main difference is that the certification phase is replaced by a weak voting phase, i.e., the delegate takes the decision to commit or abort.
Scalability capabilities of replicated databases are limited by the number of replicas, in the sense that as data volumes and number of replicas rise, operations contained in one transaction could demand accessing data from different replicas, which floods the network because of the great amount of synchronization messages that replicas have to exchange [1], so the database throughput is affected.

Because of the scalability limitations explained before, there is a new concept which brings highly scalable database at low cost. This is called NoSQL (Not only SQL) [36, 41] and is known as one of the best cloud-storage resources. In terms of scalability, the cloud offers concepts like (virtually) infinite scalability and services on demand [38].

There are some specific scenarios where the ACID [6] (Atomicity, consistency, isolation and durability) properties can be relaxed, and high performance and availability levels can be achieved. Examples of these cases are the key-value data store provided by Amazon called Dynamo [36] or the table-oriented Bigtable from Google [35]. However there are applications that cannot get the benefits of the cloud-storage due to their transactional nature (e.g, on-line sales web applications, bank applications).

The cloud offers a pay per use service [38], and for this reason it is important that cloud systems only use the amount of services that they require. This document presents a performance comparison of the aforementioned replication techniques adapted to the cloud environment. We have tried to maintain data consistency while keeping high availability and scalability.

1.1 Contribution

The aim of this work is to study the classical database replication techniques in static systems and adapt them to a cloud environment. Unfortunately, it is not possible to apply them directly, so it is necessary to partition the data previously and to distribute the resulting partitions in a subset of $M$ replicas (having $M \ll N$, with $N$ the total number of replicas).

We present a two level replication behavior. The first level consists in a small number of $P$ replicas (with $P \ll M$) which will have a strict consistency control [2] and will maintain the more recent version of the data. Each one of the $P$ replicas will propagate the changes asynchronously to the next level. We can iterate this process to several levels and, thus, we implement a replication hierarchy tree with the most recent versions at top whereas older versions at the leaves of the tree.

We provide different freshness level to each transaction, i.e. clients decide the freshness level that they need for each transaction. In this way, if a transaction has the highest freshness level it will be executed in the first level of the hierarchy, on the contrary if the freshness is low, then an older data can be provided executing the transaction in a lower level.
Finally, we have tried to execute transactions with a rational use of resources. Recall that cloud systems are pay-per-use based and we have designed a mechanism that shuts down database replicas that are not being used and starting them only when necessary.

1.2 Document organization

The rest of this document is structured as follows. Chapter 2 introduces the basic concepts and needed background of database replication. Chapter 3 details the replication technique that will be implemented and the cloud proposals currently existing. Chapter 4 explains the motivation, general description and implementation details of the proposed system. Chapter 5 presents the different experiments and their results. Chapter 6 shows the conclusions and some future work.
Chapter 2

Database Replication

This section presents the basic concepts that are necessary to understand how database replications works. Then, it explains the proposals that exist for cloud computing and databases.

2.1 Introduction

In a centralized database the serialization order of update transactions is easy to determine. But there can be different requirements for which a centralized database is not enough and it could be for two reasons: scalability and fault tolerance. In such a case, it will be necessary to distribute the database either by partitioning or replicating the data. This could provide a much better performance in a system because each client can access a nearby copy and reduce the traffic load on the network and the server. Each part or partition is managed as an individual Database Management System (DBMS) with a concurrency controller. When a partition of a database is perfect the transactions can be executed in each site without any need for communication with the rest of partitions, and the concurrency and serializability is easy to control. However the consistency management can be very difficult. The problem starts when a transaction needs data located in different partitions. This section explains the different existing options to control the transactions commit order in a replicated database.

2.2 Database Properties

It is important for a database management system to provide the ACID properties [6] (Atomicity, Consistency, Isolation and Durability). This work is based on the consistency and isolation properties for a replicated or partitioned database.

2.2.1 Atomicity

The atomicity states that database modifications must be applied all of them or none of them. All of the steps of a transaction form a logical atomic unit in the sense that it should appear to users of the database that all of these steps are carried out consecutively, without any intervening steps of other transactions.

2.2.2 Consistency

The meaning of consistency is that a transaction starts from a consistent state of the database and when it ends the database is in a consistent state. If a transaction does not satisfy this condition then it must abort.
2.2.3 Isolation Levels

Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. Isolation requires that multiple transactions occurring at the same time do not impact each other’s execution. There is a great variety of isolation levels, which differ in the level of concurrency allowed and the possible anomalies that can arise [3]. These anomalies can be avoided applying the serializability [12].

An execution is said to be serializable if it produces the same output and has the same effect on the database as some serial execution, i.e., sequentially without overlapping in time. Serializability is the major correctness criterion for concurrent transactions. However, more and more commercial database systems (e.g. Oracle, PostgreSQL, and Microsoft SQL Server) have adopted Snapshot Isolation (SI) [3] as the preferred isolation level, though it may generate non-serializable executions.

When a transaction is executed under SI it reads the data from the snapshot that was taken at the start of the transaction. This snapshot includes the most recent committed data, so read operations are never blocked. The transaction’s writes are also reflected in the snapshot and are read from there if the transaction accesses them again. If another transaction modifies the same data, only one of them can commit. To decide which transaction will commit and which one must abort exist two rules: first-committer-wins and first-updater-wins. These rules will be explained later.

2.2.4 Durability

Durability ensures that any transaction committed to the database will not be lost even with system failure.

2.3 Database Replication

When the database is replicated the consistency is oriented to maintain the data in every node with the same value at one moment in particular. This is why it is so difficult to control it, and there have been several proposals so far [5, 14, 25, 26]. The main idea is to execute the transactions in the same order in all replicas; in this way the final value of the data is going to be the same in all replicas.

One possible way to manage a replicated database consists in one database server that maintains the master copy of the database and additional database servers that maintain the secondary copies of the database [14, 26]. Database writes are sent to the master database server and are then replicated by the secondary database servers. Database reads are divided among all of the database servers, which results in a large performance advantage due to load sharing. In addition, database replication can also improve availability because the secondary database servers can be configured to take over the master role if the master database server becomes
unavailable; to accomplish this it will be necessary to provide strong consistency which consists in maintaining all copies consistent with the master database. This is a great advantage for the fault tolerance but not for performance because for every update transaction the master has to wait for a response from each secondary.

Another way to manage the replicated database is providing weak consistency. In this case it is possible that replicas diverge in the version of a given data item as the master may lately transfer the updates to the secondaries. In this way it is important to establish if it is necessary to obtain the most recent data value or it can be read from a previous one [14].

When a database system is replicated and some, or all, replicas execute the update transactions there is a trade-off between performance and consistency. If the system offers a strong consistency the performance will be degraded by maintaining updated all replicas to the latest version. A replicated database can be configured to expose inconsistent data to clients by providing a weaker consistency [27]. Some inconsistency can be found due to the propagation delay from the replica where the actual update transaction took place and its propagation. In others words if we provide a weaker form of consistency, this allows the replicated database system to achieve a bigger performance. On the other hand, a strong consistency penalizes the scalability having more replicas to maintain updated.

2.4 Replication Techniques

There are several replication techniques which present a good performance in different scenarios [5, 14, 25, 26]. A categorization of replication strategies can be done in base of two parameters [1]. The first parameter determines where updates take place. In primary copy, each object has a primary replica where all update transactions are executed and propagated to the replicas. In update everywhere all replicas accept update transactions. The second parameter determines when replicas coordinate. In an eager approach updates are executed at all replicas before a transaction commits and in a lazy approach the coordination is delayed until the transactions have committed.

2.4.1 Primary Copy vs. Update Everywhere

In the primary copy replication technique [1, 13, 15, 16] there is a replica where all transactions are executed, it is called the primary. The rest of replicas are called secondaries and they are the primary’s backups. Read only transactions can be executed in the secondaries to get the load balanced. The disadvantage of this technique is that the primary copy represents a bottleneck and a single point of failure and the mechanisms have to be designed to deal with these issues, e.g., promoting a secondary as a master. Besides, this technique lacks of scalability [13, 15, 16], since all update transactions are executed in a single replica and this compromises the performance. However, this causes some performance gain because the coordination between copies is not necessary.
The update everywhere protocols allow the update transaction to be executed in any replica. In case of a replica failure the client is forwarded to another replica. The consistency in this technique is managed globally and is much more difficult to control than in the primary copy approach since the same data can be modified by several transactions in different replicas. All the updates must be executed in all replicas, so it is necessary to control the order in which they are executed.

2.4.2 Eager vs. Lazy

Eager Replication updates all replicas in one atomic operation keeping all copies synchronized. This generates a considerable amount of messages and, thus, reducing the performance and increasing the transaction response time; since the user does not receive the answer until sufficient copies in the system have been updated [1, 15, 16].

Lazy Replication [1, 15, 16] propagates the updates to all replicas in an asynchronous way, having much faster response time for the transaction but less consistency as opposed to the eager approach. The update is done in the local copy, and some time after it propagates the changes. All these techniques can be combined as shown in Figure 2.1.

2.4.3 Eager Primary Copy Replication

In this approach [1, 15, 28] an update transaction is executed in the primary copy and then it is sent to the secondary replicas. The primary copy waits until all secondary copies send the commit or abort response, if all responses were “commit” then the transaction will commit, or, otherwise, abort and the response is sent to the client. Read only transactions are executed in any copy; recall that the transaction will receive the most recent version of the data as all replicas are updated synchronously. The ordering of transactions is made in the primary copy and the rest of copies must follow it. This combination has the benefits of the eager replication and in the case of primary copy failure any secondary can take its place as it has the most recent version; however the primary copy is always a bottleneck.
2.4.4 Eager Update Everywhere

In the update everywhere technique the update transaction can be executed in any replica. This is why it is so important to control the order of the transactions globally. To combine these two techniques it would be necessary to apply a distributed locking and a total order broadcast [15, 18, 28]. This will be discussed later.

For the distributed locking the data can be accessed only when they are locked at all replicas. When a server receives the request of the transaction it sends the lock request to the rest of replicas, and it waits until it is blocked at all replicas. If the lock is granted the transaction is executed if not, the transaction is delayed and the request is sent later. After the transaction is executed in all replicas the client receives the response [12].

With the total order broadcast communication primitive a message can be sent to the replicas with the guarantee that all replicas agree on the set of messages delivered and the order according to which the messages are delivered [17]. In the case of a replicated database the client submits the request to one replica referred to as delegate to broadcast the request to the rest of replicas and is coordinated by the order given by the total order broadcast. In case of conflict between operations, they are executed in the order indicated by the atomic broadcast. Once the transaction is executed the answer is sent to the client by the delegate replica.

2.4.5 Lazy Primary Copy Replication

In this case all clients must connect to the same replica (the primary) to perform an update. The result of combining lazy replication with the primary copy is that the transaction is executed in the primary and the response is sent to the client before the response from the secondaries is received [15, 28].

2.4.6 Lazy Update Everywhere Replication

There is no much difference between this technique and the previous one. Sometime after the commit the updates are propagated to the rest of replicas. However coordination can be complicated since the other replicas could have run a conflicting transaction at the same time and this could lead to an inconsistent state [15, 28]. To solve this, a reconciliation process is needed to decide which transaction must be undone. Efficient reconciliation processes have been proposed in [5, 25, 29, 30, 31].
2.5 Replication Techniques Based on Total Order Broadcast

There are other techniques which are based on total order broadcast [2]. These techniques have in common that they are update everywhere replication and they require $\Theta(1)$ network interactions. The most relevant techniques are: active replication, certification-based replication and weak voting replication [2].

2.5.1 Active Replication

In this case all the transaction is put into a message and broadcast to all servers using the total order broadcast. The client does not concern for the total order broadcast, instead the delegate server does. The principal disadvantage is that all operations (read and write) are sent in the transaction, so all the readings are executed in all servers losing one of the benefits of the replication: load distribution for read operations [2].

---

A technique is said to have $\Theta(1)$ network interactions, if the number of interactions between servers in order to process a transaction is constant, i.e., independent of the number of operations in the transaction.
Figure 2.2 shows the communication scheme of this technique. When the delegate server $s_d$ receives the transaction $t$ from a client $c$, the server $s_d$ broadcasts the transaction to the rest of replicas using total order broadcast. All servers delivered and process $t$. Each server serializes the transaction in the same position to make it deterministic, if one server aborts then the rest must abort. Figure 2.3 shows the pseudocode for this technique.

### 2.5.2 Certification-Based Replication

This technique executes a certification phase that decides if a transaction commits or aborts [2]. When a transaction $t$ is received by a delegate server $s_i$ from a client $c_i$, $s_i$ executes it but delaying its write operations. When transaction issues its commit operation, the writeset (the set of updated items) and readset (the set of read items) are sent to the rest of servers using the total order broadcast. Upon delivery of $t$, each server executes a deterministic certification phase. In this technique the transaction is processed and can be aborted in the certification phase due to a conflict. In the certification phase $t$ is validated against concurrent transactions looking for write-write and read-write conflicts. Since all replicas can hold the same history list of previously delivered readsets and writesets, no additional communication step is needed for that [5, 25, 32]. This technique is optimistic and is very effective in conflict scenarios. This technique is effective in low-conflict situations. All the process is illustrated in Figure 2.4 and the pseudocode can be found in Figure 2.5.

![Figure 2.4. Certification-based replication schema](image)
2.5.3 Weak Voting Replication

This technique is very similar to the certification-based replication. The main difference is that there is no certification mechanism but a weak voting phase. In this phase the delegate server determines whether another conflicting transaction has been previously committed and makes the decision if the transaction can commit or must abort. This technique uses the total order broadcast.

![Diagram of Weak Voting Replication Scheme](image-url)
Figure 2.6 shows the schema of this technique which works as follows: when the delegate server $s_d$ receives a transaction $t$ from client $c$, the delegate server executes $t$ but delays the write operations. When commit time is reached the transaction is broadcast to all servers using total order broadcast. Then, $s_d$ determines if any conflicting transaction has been committed. Based on the result of this, the delegate server sends a new broadcast indicating the result of the transaction. See Figure 2.7 for the pseudocode of this technique.

### 2.6 Isolation Levels in Database Management Systems

Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions.

To apply the isolation levels, the database management system must apply some control in order to define whether locks are taken when data is read, the type of locks, how long the read locks are held. In case that a read operation refers to rows modified by another transaction it can block until the exclusive lock on the row is freed. Another option would be to retrieve the committed version of the row that existed at the time the transaction started or to read the uncommitted data modification.

A lower isolation level increases the ability of many users to access data at the same time, but increases the number of concurrency effects [3]. These effects could be caused by different scenarios; e.g., lost updates occur when two or more transactions select the same row and then update the row based on the value originally selected. Each transaction is unaware of the other transactions. The last update overwrites updates made by the other transactions, which results in lost data. Another concurrency effect could be caused by reading an item from a second transaction that is being updated by another transaction. The second transaction is reading data that has not been committed yet and may be changed by the transaction updating the row. If the
second transaction accesses the same row several times and reads different data each time this is called a Non-repeatable read.

On the other hand, a higher isolation level reduces the types of concurrency effects that users may encounter, but requires more system resources and increases the chances that one transaction will block another. Choosing the appropriate isolation level depends on balancing the data integrity requirements of the application against the overhead of each isolation level. The highest isolation level, serializable [3], guarantees that a transaction will retrieve exactly the same data every time it repeats a read operation, but it does this by performing a level of locking that is likely to impact other users in multi-user systems. The lowest isolation level, read uncommitted [3, 14, 21], may retrieve data that has been modified but not committed by other transactions. All of the concurrency side effects can happen in read uncommitted, but there is no read locking or versioning, so overhead is minimized.

Serializability has been accepted as the appropriate notion of correctness for executions of a collection of transactions against a database [12, 14, 21, 23]. A serializable execution is one in which each committed transaction sees the same values, and the final state contains the same values, as in a serial or batch execution, where the transactions are run one-by-one with no concurrency at all.

2.6.1 Snapshot Isolation

There is a concurrency control which avoids taking shared locks, by accessing versions of data [21]. This is called Snapshot Isolation (SI), it avoids all the concurrency anomalies such as Inconsistent Read and Lost Update [3]. However, as proved in [3], this algorithm does not ensure serializable execution for all transactions and it can lead to violation of integrity constraints. In simplified terms, the Snapshot Isolation concurrency control mechanism means that a transaction T sees the database state produced by all the transactions that committed before T started. Thus if T₁ and T₂ are concurrent transactions using the Snapshot Isolation mechanism, neither will see the effects of the other (whereas in a serial execution, and therefore also in a serializable execution, one of the transactions would see the effects of the other).

However, Snapshot Isolation (SI) is used in several DBMSs. This technique allows that a read-only transaction never causes blocks or aborts to update transactions and they never block or abort. This improves the performance of applications which are mostly made of read-only transactions (e.g. web applications).

In the definition of SI [3], the system assigns a transaction t a start timestamp, called start(t) when the transaction starts; this determines the database state seen by t [14, 23]. Updates made by any transaction t₀ that commits after time start(t) will not be visible to t. Updates made by any transaction t₀ that commits before start(t) will be visible to t. SI also requires that each transaction t be able to see its own updates. Thus, if t updates a database object and then reads that object, it will see the updated version, even though the update occurred after start(t).
When a transaction $t$ commits, it is assigned a commit timestamp, $\text{commit}(t)$, where $\text{commit}(t)$ is more recent than any start or commit timestamp assigned to any transaction [3, 4, 14]. $t$ commits only if no other committed transaction $t_0$ that is concurrent with $t$ wrote data that $t$ has also written. Otherwise, $t$ is aborted so as to prevent lost updates. This technique for preventing lost updates is called the first-committer-wins (FCW) rule [14, 22, 24]. If $t$ successfully commits, then any transaction $t_0$ that is assigned $\text{start}(t_0) > \text{commit}(t)$ will see the updates made by $t$. There is another rule called first-updater-wins [22] that has the advantage of simplifying an application’s commit logic, but the disadvantage of being subject to greater risk of deadlock. However, SI retains the benefits of writers not blocking readers. This rule is implemented by PostgreSQL. Both rules are similar and have the same effect, they abort one of the concurrent transactions. The difference between these two rules consists in the validation that they apply. The first one makes the validation when the transaction commits while the second one validates at the update moment and this allows to roll back the transaction earlier and not at commit time.

All this is easy to apply in a centralized environment. When we move to a replicated database, it is necessary to aggregate other characteristics to maintain consistency. The problem comes from the requirement that a transaction must see the more recent snapshot at the beginning. To implement the order to define the latest snapshot and make the corresponding snapshot available may cause delays and the loss of one of the benefits of the Snapshot Isolation.

In serializability theory, a replicated system is 1-copy-serializable (1CS) if the execution in the replicated system is equivalent to a serial execution on a centralized database [5]. 1CS presents some drawbacks: read operations can be blocked; which is a big problem for dynamic content web page generation. As a consequence of this and because most of commercial and open-source databases provide Snapshot Isolation (SI) [3], the GSI correctness criterion is used.

To adapt the SI to a replicated system, GSI [4, 33] is used. This is based on the observation that a transaction can obtain an older snapshot but the rest of SI properties are maintained. In particular, the FCW rule for update transactions is maintained. Some conditions can be identified to guarantee 1-copy-equivalence execution. When an update transaction commits, its writeset must be checked with the writeset of any committed transaction. If another transaction has written on the same data from the moment the snapshot was taken.

Snapshot Isolation and Generalized Snapshot Isolation have two rules [4] in common: the first ensures that each transaction only observes a committed snapshot. This could be any snapshot taken before the transaction starts. The second rule prevents a transaction from committing if it is impacted by another committed update transaction. To decide this, a test called certification is made. In this test the database checks if the writeset of the transaction intersects with the writeset of any committed transaction.

In [4] can be found an explanation of serializability under GSI. To accomplish this, a third rule is added. This rule is dynamic and requires that the readset does not intersect with the writeset of the same set of committed transactions. There is a static condition which avoids the
need of the readset to certify an update transaction. If the writesets and readset of the transaction can be determined statically, then the condition explained before can be applied statically rather than dynamically. To use the static condition the update transactions must be known.

2.7 Cloud Computing proposal

Cloud computing is a term that is used to refer to a considerable amount of hardware resources which are available online [39]. It is service-oriented. According to the provided capability, the services of cloud computing can be divided in three categories: Infrastructure-as-a-Service (IaaS) offering processing, storage and network; Platform-as-a-Service (PaaS) supports a set of application program interfaces to cloud applications; and Software-as-a-Service (SaaS) replaces the applications running in the local PC [38]. Another characteristic is elasticity, users can have as much or as little of a service as they want at any given time; and, the service is fully managed by the provider, for these reasons it is a low cost solution from the application developer point of view. This works for those systems which have a large amount of data or need a high availability and scalability.

The CAP theorem [7, 34] establishes that it is impossible to guarantee in a system these three properties: Consistency, Availability and Partition tolerance. In a partitioned system the “P” part has to be ensured and for this reason it seems to be a need a choice between Consistency and Availability.

It is known that cloud storage systems do not provide full transactional support [35, 36]. However, there are several proposals of storage systems for the cloud that support transactions. One of them is ElasTraS [8] an Elastic TranSactional relational database, which is fault-tolerant, self-managing and is designed to scale out using a cluster of commodity machines. This system executes the transaction in main memory having one thread per partition.

There are some characteristics that every system aimed for the cloud needs and were included in the design of ElasTraS [8]:

- Scalability. A DBMS deployed in the cloud should be able to scale.
- Elasticity. The systems deployed onto the cloud exploit the elasticity that is offered by expanding or shrinking the resources that they use considering the load.
- Fault-Tolerance. A system for the cloud should be designed to support failures and remain operational.
- Transactional Access. As a RDBMS has to provide atomic, consistent and durable access to data, it is desired for a cloud DBMS to maintain this.

As part of the design the system has an application state (actual data stored) separate from the system state (Meta information critical for the functioning of the system), which allow the use of different protocols to manage the different components in the best way.
ElasTraS uses a light locking granularity to allow the loose of union between the nodes having a light synchronization mechanism that scales to a large number of nodes. Time leased is used to ensure mutual exclusion of critical operations and support node failures. Besides, the operations are limited to a single node allowing the horizontal partition of the load. This gives a big improvement and this is that if a component fails, this does not affect the operation of the other components.

The technique applied by ElasTraS to manage the database consists in partitioning the data at a schema which allows to locate in the same partition the related data fragments from multiple tables. This technique improves the performance of transactions by reducing the number of partitions that must be accessed.

A description of the ElasTraS architecture can be found in [8]. It is composed by a distributed fault-tolerant storage which allows to decouple storage of data from the ownership. If a decoupling of data and ownership is made the data movement is minimized. If it is necessary to move a partition for fault tolerance, load balancing or elasticity, it would be much easier if the ownership is separated from the data. Another component called Owing Transaction Managers (OTM) owns one or more partitions and provides transactional guarantees on them. A third component is the monitor of the OTMs, and is responsible for recovering from node failures, performing load balancing and elastic scaling and is called TM Master. The fourth component is the Metadata Manager (MM). It comprises a mapping of the database partitions to the owners, and the leases. Upon all these components, there is a Client Library which provides a wrapper for the system and hides the complexity of locating partitions.

ElasTraS is based on the philosophy of key-value stores to minimize distributed synchronization and remove scalability bottlenecks. It is designed to scale up and down with the transaction workload as explained in [8]. This system has several limitations because it does not provide transaction recovery and it just supports a restricted transaction semantics that is executed in one data partition.

There is another system created for the cloud, it is called EcStore [10] and it is an elastic cloud storage system that supports automated data partitioning and replication, load balancing, efficient range query and transactional access. It has three layers: a distributed storage, replication and transaction manager. This allows to implement a combined design of multi-version and optimistic concurrency control. In this schema each transaction has a timestamp at the beginning and at the moment of the commit. When a transaction accesses a data object, it sees the most recent version of the data with the timestamp less than transaction’s timestamp.

EcStore [10] provides Snapshot Isolation applying a protocol where all transactions at the commit time are validated. Each transaction at commit time is validated against other transactions successfully committed, except for the read-only transactions which are executed in a consistent snapshot of the database and they do not need the validation phase. To avoid the bottleneck in the primary copy the read-only transactions access to the replicas, but the update transactions access to the primary copy to ensure that the updates are well-behaved. All the characteristics
presented give to this proposal load-adaptative replication and an optimistic concurrency control which have an efficient cloud storage balancing the load distribution under a big amount of workloads.

Google Megastore [37] is a layer placed on top of a key-value database (concretely, Bigtable [35]) with the aim of accepting regular ACID transactions with an SQL-like interface in a highly scalable system. It is a storage system developed to meet the requirements of interactive online services. It uses synchronous replication to achieve high availability and a consistent view of the data. The data model is declared in a schema. In Megastore the data is partitioned and then replicated to each partition separately, providing full ACID semantics within partitions, but only limited consistency guarantees across them. Data for most Internet services can be properly partitioned, and a small set of features can substantially ease the charge of developing cloud applications. The replication system in Megastore provides a single, consistent view of the data stored in its underlying replicas. Reads and writes can be initiated from any replica. Replication is done synchronously, replicating the transaction log to the replicas. Writes require one round of replica communication, and reads run locally. A read always gets the most recent version.

In the cloud, there is a way to manage the system called Database-as-a-Service (DBaaS) that provides the benefits of working with resources that are controlled by a third part. The MIT based on this service propose the Relational Cloud [11]. That system is based on three topics that allow to adapt it to the demands that cloud storage systems have. First, the system minimizes the amount of hardware required for a workload. Second, it has elastic scalabity tolerating the growing of the workload and balancing it to several machines. Third, it provides privacy which is very low for the system on the cloud. For this system, it is accomplished by encrypting the data.

The partition of the database is made looking the way to minimize the workload in the nodes of the database. In this way, it would be easier to scale if the database can be partitioned into new nodes but having the less amount of multi-node transactions. Another important characteristic is the capacity of relocating the database partitions through different nodes. This makes easier to maintain and administrate the database and to manage the load changes when new nodes are added or removed. This system is still being developed but it presents very interesting options to manage data making it scalable and available which, as it was showed before, are two basic characteristics for the cloud.

Another cloud system is Deuteronomy [9] which supports efficient and scalable ACID transactions in the cloud. To achieve this, the operations are decomposed into a transactional component that manages transactions and their concurrency control, but knows nothing about the physical data location; the second component is a data component which maintains a data cache and uses access methods to support a record-oriented interface with atomic operations. Deuteronomy distinguishes itself from previous works by its approach of factoring the functions into the two components previously mentioned.

The Deuteronomy architecture is scalable in three dimensions. From a user perspective, if more application execution capability is needed then more application servers can be added. If
data volume grows, more data components can be added underneath a transaction component to handle the storage and data manipulation workload. For the case that transaction rates reach a degree that saturates the computational resources of a single transaction component, more transaction components can be instantiated supporting transactions on disjoint sets of data.
Chapter 3

Replicated Database System Design

3.1 Analytical Model

This chapter describes the system proposed in this work. It simulates the replication techniques previously explained combined with a cloud scheme to provide high scalability, availability and an appropriate use of resources. Section 3.1.1 explains the motivation and Section 3.1.2 explains the system model.

3.1.1 Motivation

It is well known that the performance of replication protocols depends on the workload characteristics. For instance, the primary copy can provide a higher throughput in a replicated system where most transactions are read-only [14]. On the other hand, if there is a lot of update transactions, an update everywhere schema based on total order broadcast may be more appropriate [2].

However, lazy techniques give a fast response to the client because the transaction is executed in the delegate replica and it is not necessary to wait for its propagation to the rest of replicas. However, they offer poor consistency; different replicas can have different versions of the data. On the contrary, the eager techniques can show a higher response time but maintain higher levels of consistency [2].

Techniques based on update everywhere present low scalability because of the costs of propagating the changes made by transactions in total order broadcast and applying them in remote replicas. This scalability disadvantage can be alleviated by creating a hierarchical server structure for update propagation. In this case, an update everywhere protocol would be combined with a primary copy protocol. Let us say that there are $S$ servers (one database partition per server), and each one has $N$ children. These $S$ servers receive the transactions from the clients, each server behaves according to the update everywhere protocol implemented and the response is sent to the client. Then, the transactions are sent by these servers asynchronously to each child, having a primary copy behavior. And this can iteratively go on so that it could exist as many hierarchical levels as necessary. This causes that at a particular moment the replicas in the lowest level of the hierarchy have an older version of the data and the lower is the level in the hierarchy the older is the version as shown in Figure 3.1.
At the beginning (Figure 3.1.a) the partition has an initial version \(v=0\); afterwards, a client executes an update transaction that modifies the initial version and the partitions in the next level are updated (Figure 3.1.b). In the same way, each replica will propagate the changes to its associated secondaries (Figure 3.1.c) that respectively propagate the changes to their associated replicas (Figure 3.1.d). Meanwhile, the client can execute another update transaction and modify the partition state (Figure 3.1.e). As a result, the partition can have different versions of the same data item along its associated replication depth (Figure 3.1.f).

When a transaction is received it can be forwarded to a secondary replica depending on the consistency level required, achieving load balance. A further advantage of the presented architecture is that system replicas can be upgraded or downgraded in the hierarchy according to current system requirements.

### 3.1.2 System Overview

Bearing in mind the previous ideas, we propose to create a simulator that implements the replication techniques explained in this work. Recall that these replication techniques were
evaluated in [2] and the results obtained assumed a static system. For this work we propose to evaluate its performance for a cloud system having as the dynamic part to promote or degrade the replicas to achieve the load balance, as well as to allow executing the read only transactions in a replica located in a lower level in the hierarchy explained depending on the data freshness required. Furthermore, we propose to turn off or turn on the replicas to consume the lowest amount of cloud resources.

![Figure 3.2 Proposed structure for the servers](image)

The system consists in a partially replicated database. To this end, it is necessary to have a component which controls the database partitions and creates the hierarchy mentioned that results in the structure shown in Figure 3.2. The servers in Level 0 take the behavior specified by the replication technique implemented (one of those introduced in Chapter 2). To the rest of levels the protocol applied is primary copy to transfer the changes in an asynchronous way. In case that a client sends an update transaction $t$, the transaction is executed by a replica in Level 0 managed by the replication technique implemented. After all the process is completed and the client receives the response, the changes are propagated to the replicas in Level 1 (lazy primary copy protocol). In the same way, the replicas in the Level 1 will propagate the changes to the replicas in Level 2 asynchronously.

With this structure, it is possible to have different versions of the same data in one partition. In this way, the read-only transactions can be forwarded from the replica in Level 0 to one replica in Level 1 or Level 2 as the freshness required by the transaction is lower. Figure 3.3 shows this behavior, a client sends a transaction $t$, the freshness level is the highest so it has to be executed at Level 0. A second transaction is sent but the freshness level is lower so it is executed in Level 1. All the changes are propagated to the rest of levels.
As it was explained before, cloud services are pay-per-use based; it means that the client only pays for consumed resources (platform, software or infrastructure). Because of this, it is
important to have systems that can consume the least resources as possible providing good performance. To resolve this, we propose to shutdown the servers that are not used. As it can be seen in Figure 3.4, initially all the servers in Level 0 are active (Figure 3.4.a). When one server in Level 0 reaches its maximum capacity, sends a notification to the metadata manager (MM) (Figure 3.4.b) and one of the servers at Level 1 is promoted (Figure 3.4.c) at the time it receives the notification from the MM. First, the server must be updated, so it has the most recent version, and then it can execute the transactions sent by clients. The transactions with the strict freshness degrees can be executed in it, and the load of Level 0 gets minimized. If this server reaches the maximum of its capacity (Figure 3.4.d) then one of its children is updated and activated. If the working load in Level 0 keeps increasing, then another server in Level 1 will be started (Figure 3.4.d). If a server has no children, then a server from its level is promoted. On the contrary, when the active servers in the higher levels reach their minimum capacity they are shut down, and resources are released as is shown in Figure 3.4.e. A server in Level 1 reaches its minimum capacity, so it sends the notification to the MM to deactivate the server in Level 2 and so on. In Figure 3.4 a new component can be observed. This is the MM which is in charge of the replication of the database previously mentioned and controls the replica upgrade or downgrade process. This will be explained in the next section.

3.1.3 System Model

The created system is composed by three main components:

- **Client Applications.** Clients are sources of transactions. A client submits the transaction and receives the response from the server. These events are repeated following certain parameters and are controlled by a timer. A client only can submit one transaction at a time. One server is connected with many clients. Each server is connected with the same number of clients. Clients gather all the performance data and compute statistics.

- **Servers.** Each of them store one database partition. Each server hosts a local database manager with a replication technique running on top of it. To accomplish this, the server application is structured in three levels:
  1. Communication module, that represents the interactions among servers and each server with its associated clients. There is one instance per server. This module allows the server to send multicast messages and manages the multicast group to which the server will communicate. Implements the primitives to the total order broadcast. Also controls the clients that are connected to the server. The message queues in each node are FIFO.
  2. Database module simulates a database system. There is one for each server. This module includes a lock manager and a data manager. The lock manager supports all the logic needed by each one of the replication techniques explained in section 3. The data manager handles operations to read and write data from/to the database. We assumed that all the data is in main memory.
3. Database replication module. It manages the database replication technique. There is one instance of this module for each server. The module behavior depends on the replication technique used in the simulation. Each of the techniques explained in section 2 has been implemented.

- **Metadata manager.** It controls the system state and the communication with clients and replicas. This module is in charge of the initial database replication. It monitors the resource utilization, allocating the data partitions and controls the correct propagation of updates. When a server reaches the maximum capacity notifies the metadata manager, so a child could be promoted. In the same way, when the server reaches the minimum load, the metadata will turn off one child at a time to save resources.

The mentioned components and the interaction between them are shown in Figure 3.5.

![Figure 3.5 System Model](image-url)
Chapter 4

Simulation System Specification

We have developed a simulator using version 2.34 of the NS2 network simulator [40] which was created using C++ code and otcl as scripting language to accomplish the performance analysis of classic database replication techniques in a cloud system. The simulator runs according to a set of configuration parameters, which are specified in an XML file. This chapter explains the specifications for the simulator.

4.1 System Specifications

We assume a total of $P$ partitions, each one with a replication degree of $R$; thus, the total number of servers $S$ in the simulation are $S = P \times R$. Each server will be denoted with the expression $s_n$. The clients will be denoted by $c_i$. Each server $s_n$ has the same number of clients $c_{S_n}$, which makes the total number of clients $C = S \times c_{s_n}$.

The set of all possible transactions is $T$. A transaction sent by the client $c_i$ will be denoted as $t_{ij}$. Each transaction $t_{ij}$, contains a set of operations that must be executed. These operations can be writes and reads. A client $c_i$ has a specified freshness parameter (it indicates the version tolerance for data) that will be represented in the transaction sent as $t_{ij}.freshness$. Finally the transaction must specify if it is read-only or not with the parameter $t_{ij}.readOnly$. The result of the transaction $t_{ij}$ sent by the server is $result_{ij}$.

4.2 Client Specifications

Clients are processes that generate transactions, send them to the servers, wait for response, and, when the response is received, they process it and collect the statistics. Once the transaction is sent by $c_i$, it waits to receive the response to send the following transaction. The client $c_i$ knows its delegate server in advance and if this server reaches its maximum capacity, the client will get notified and connect to a new server.

Every client $c_i$ has a very important parameter, the time between transactions $d_i$ and it is configurable. This parameter represents the time between the start of two transactions, which means if a client $c_i$ starts a transaction at time $tm_i$, once the transaction is finished, the client $c_i$ waits until time $tm_i + d$ to generate the next transaction.

4.2.1 Client pseudo-code

Figure 4.1 shows the pseudo-code on the client. This code is executed by all clients. At the beginning the variables are initialized, the freshness variable is obtained from the configuration file (line 7). Transactions are generated in the generateTransaction task (lines 9-17). This is executed following the interval specified in the configuration file. In this task, the transaction is created and
the operations are assigned, as well as the identifier in the client, and it is forwarded to the server (line 16).

When the client receives the response from the server it processes the response verifying the result (line 21) and updates its statistics accordingly (result and response time). See line 24 of Figure 4.1.

![Figure 4.1 Client pseudo-code](image)

### 4.3 Server Specifications

Each server \( s_n \) receives the transactions generated by its delegated clients. The server must keep track of each one of the transactions received so it will know to which client they belong to and notify the proper client. Each server manages a version number. For each update transaction that commits this version gets incremented. This information (updates and version) is propagated to its associated children in the replication hierarchy if needed. The replication process through the hierarchy continues till the replication depth; although we have fixed it to three.
4.3.1 Communication module

This module has a list of clients connected with the server $s_n$, a list of the rest of servers in the simulation and a list of servers which have the same database partition (replicas). Each one of these lists is accessed whenever a message is sent.

![Figure 4.2 Communication module pseudo-code](image)

Figure 4.2 shows the pseudo-code for a communication module. When a server $s_n$ receives a transaction one of the following scenarios will take place: (1) the client $c_i$ sends the transaction $t_{ij}$ it is received by the communication module in server $s_n$ (line 7) as a local transaction and this module notifies the server that a transaction has been received (line 9). (2) The transaction $t_{ij}$ is received from another server $s_m$ (task recvTransFromServer in line 12). This means that the transaction is remote and has to be applied, or not, depending on the replication technique being considered. (3) The transaction $t$ is received from a replica in a higher level, then it will be treated differently as it is a writeset-version pair of data, and the server $s_n$ knows that it must apply an update (see task in line 17).

After the execution process is finished, the server must send the response. In the case of a local transaction, once the server has the response it is sent to the client through the communication module (line 22). When the transaction is remote, the response is sent to the corresponding server using the sendResponseToServer task (line 32). When the sendUpdate task (line 28) is invoked by the server $s_n$, the communication module sends the different versions with its corresponding writeset to update the children using a multicast message with the sendMulticastChildren task (line 39).

The sendMulticast task in line 25 sends the transaction to the rest of replicas at the moment that it is invoked by the server. Depending on the replication technique applied, if it receives an
abort message, then the rest of replicas must be notified so they abort transaction \( t \) as well, this is done in the sendNotification task (line 34 of Figure 4.2).

### 4.3.2 Database module

The modules presented so far did not explicitly depend on the replication technique; this module does depend on it. The database module simulates a database system and has two components: the lock manager and the data manager. Figure 4.3 shows the pseudo-code of the lock manager and data manager.

![Figure 4.3 Pseudo-code for lock manager](image)

When the communication module notifies that a transaction \( t_{ij} \) has been received, the server \( s_m \) invokes the database module, sending the transaction \( t_{ij} \) to be executed. The database module passes \( t_{ij} \) to the lock manager so it will be added to the transaction FIFO queue invoking the addTransaction task (line 5). As the nextTransaction task in line 10 is invoked, this task verifies if there is a transaction in the queue, if so, and then it is executed, otherwise, waits to be notified for an incoming transaction.

Finally, when a transaction is taken from the queue and is about to be executed, it first verifies if the transaction has no conflicts at all. This is done in the conflict task (see Line 1 in Figure 4.3). If the task does not detect any conflict, the transaction executes; if there is a conflict then the server is notified and the transaction aborts.
4.3.3 Database Replication Module

This module implements all the replication techniques explained though only one is executed per partition. There is one instance per server, and it decides what to do next in the execution process. For instance, it gives the order to the server to answer to the client or send the transaction to the rest of replicas (eager or lazy), and it applies the certification phase if needed. The Figure 4.4 shows the pseudo-code for this module. At the beginning the protocol to be applied is read from the configuration file at the initialization moment. Then, each server calls the replicationProtocol task in line 4 and depending on the protocol, one of the tasks is executed (lines 8-26).

![Figure 4.4 Pseudo-code for Database Replication Module](image)

4.3.4 Pseudo-code of the server

The server has three components (communication module, database module and replication database module) and they are unified inside the server. To this end, it has the pseudo-code that is shown in Figure 4.5. The server $s_n$ receives a transaction $t_{ij}$ from the client $c_j$, and as it was explained, the server receives a notification from the communication module through the task receiveTransaction in line 4. Then, $s_n$ sends the transaction to the database module so it could be executed by the addTransaction task. After this, $s_n$ receives a notification with the result of the transaction from the database module through the response task (see lines 16-22) so it can send the response to the client.
In case transaction $t$ is received from another replica $s_m$ (line 8), this transaction is considered remote and must be executed according to the replication technique applied and the response must be sent to the server $s_m$ (line 19). When the server $s_m$ receives the response of the transaction $t$ from replica $s_n$, then it must be processed (see task in line 23). In this way, the server must have the control of transactions and their responses so when it reaches the total number of replicas, it can notify the client if an eager technique is used or tell the rest of replicas that the transaction has been aborted (sendNotification task from the communication module).

Finally, when the server $s_n$ has to update the children it invokes updateChildren (line 28). When the children receives an update from the father (line 11), they must take the data sent and apply the write operations and update the version, as the children could receive several messages of this kind, they must repeat this process for each version received.

Figure 4.5 Pseudo-code for server
4.3.5 Metadata Manager Module

This module knows all servers in the simulation. It controls the database partition and replication, assigning the partitions to the servers. Figure 4.6 shows the pseudo-code for this module. It has the control of the server structure, so when a server has reached its maximum workload capacity, it notifies to the metadata manager and the MM orders to one of the servers in the next level in the hierarchy to promote.

```
1. task initialization
2. servers <- S
3. end task
4. task partitionDatabase
5. foreach s_i in S do
6.   s_i <- database partition
7. end foreach
8. end task
9. task promoteChild
10. when s_i maximum then
11.   foreach child in s_i do
12.     if child is not promoted then
13.       active child
14.   end if
15. end foreach
16. if all children were promoted then //if all the children are promoted then look in
17.   look in the next level //the next level
18. end if
19. end when
20. end
21. task backToLevel
22. when s_i reach a minimum load then
23.   foreach child in s_i do
24.     if child is promoted then
25.       deactivate child
26.     break
27. end if
28. end foreach
29. end when
30. end task
```

Figure 4.6 Pseudo-code for Metadata Manager

The `partitionDatabase` task takes the database and makes the partition according to the configured parameters. Afterwards, these partitions are assigned to servers.

When a server reaches its maximum workload, it notifies to this module in the `promoteChild` task (see line 9 in Figure 4.6). The MM looks into the children of the server, and promotes one of them to the previous level in the hierarchy. When a server reaches its minimum load level, notifies it to the metadata manager that returns this server to its immediate successor level.
4.4 Replication Techniques Specifications

Recall that there is only one replication technique executed at a time. This section details the tasks in lines 8-26 from the pseudo-code of the database replication module in Figure 4.4.

4.4.1 Eager Primary copy

After a server $s_n$ receives the transaction $t_{ij}$, it is executed in the database module and just before the commit is sent to the replicas through the communication module (it only sends the writeset). Replica $s_n$ waits for all the other replicas responses. If all were commit, then it commits and sends the response to the client; otherwise it aborts.

4.4.2 Lazy Primary copy

This protocol is very similar to the previous one, the main difference relies on the moment of the update propagation. In this case, the delegate copy executes the transaction in the database module and the response is sent to the client in the response task of the server; after this, the writeset is sent to the rest of replicas using the sendMulticast task of the communication module.

4.4.3 Update Everywhere

This protocol has three replication techniques that were developed in this work:

Active Replication

This technique is based on total order broadcast. The server $s_n$ receives the transaction $t_{ij}$ from client $c_i$. The server $s_n$ sends the transaction $t_{ij}$ (writeset and readset) to the rest of replicas invoking the sendMulticast task from the communication module; in this case the total order broadcast is applied. Each replica processes $t_{ij}$ in a deterministic way. So each replica serializes the transaction in the same position.

Certification-based replication

This technique is similar to the previous one. The difference is that when the commit time is reached in the delegate server $s_n$, the transaction $t_{ij}$ is sent to the rest of replicas using total order broadcast in the communication module. The transaction $t_{ij}$ is received in the replicas, and a deterministic certification phase is executed in the database module which decided if transaction $t_{ij}$ can commit or abort. The rest of the process is the same; the replicas response to the delegate server $s_n$ and this response to the client, respectively.

Weak voting replication

In this case the server $s_n$ sends a multicast message twice, the first one is after all the receive process and execution of $t_{ij}$, as it has been explained before; the server $s_n$ sends the writeset of the transaction using the total order broadcast primitive from the communication
module. Then, the server \( s_n \) can determine if the transaction commits or aborts based on the conflicting transactions committed provided by the database module. Based on this information, \( s_n \) sends another multicast through the communication module to the replicas, indicating the result of \( t_{ij} \). This result is received by the communication module in each replica and notifies to the server \( s_m \). When the server replica receives the result it processes it and executes the transaction in the database module knowing the result.

### 4.4.4 Eager update everywhere

Transaction \( t_{ij} \) is sent by the client \( c_i \) to server \( s_n \). Once \( t_{ij} \) is received and executed by \( s_n \) in the database module through the `executeTransaction` task, and just before its commit is reached, the writeset in the transaction \( t_{ij} \) is sent to the rest of replicas using the `sendMulticast` task from the communication module. \( s_n \) waits for all responses, as the responses from the replicas arrive they are processed in the `receiveResponse` task in server; after all the responses are received, \( s_n \) sends the commit or abort to the client using the `response` task in the server.

### 4.4.5 Lazy Update Everywhere

This technique is similar to the previous one, the main difference is that the server does not wait for the replica responses to execute the transaction received. In this case, the transaction received \( t_{ij} \) is executed in \( s_n \) and the response is sent to the client \( c_i \) invoking the `sendResponse` task in the server. After this, the `sendMulticast` task is invoked so the writeset is send to the rest of replicas. The process to update the children is the same explained in the previous technique.

### 4.5 Client-Server communication

The transactions are generated in the client according to some configured parameters. The transactions are received by the corresponding server (the one connected to the client). The server processes the transaction according the replication technique applied, sending the complete transaction or just the writeset to its replicas and its children. Once the transaction is executed, or all responses are received, the response is sent by the server to the client. When the client receives the answer, it processes the result and completes the statistics of abort rate and response time.

### 4.6 Implementation Details

All experiments were carried out using a discrete event simulator targeted at networking research called NS2 [40]. It was used as an underlying layer to networking communication. NS2 is an object oriented simulator, written in C++, with an OTcl interpreter as a frontend. The simulator supports a class hierarchy in C++, and a similar class hierarchy within the OTcl interpreter.

Knowing this, the simulator was developed using these two languages. The replication protocols were implemented in C++. This requires an efficient manipulation of byte sequences,
and packet headers. With this, the algorithms that support the database replication behavior have been implemented. On the other hand, OTcl was used to develop the network creation and configuration.

C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly, making it ideal for simulation configuration.

Different classes were added to NS2 to adapt it to a replicated database system simulator. These classes implement all the logic explained in the previous section.

The clients and the servers are nodes in the simulator; these nodes are composed by the modules provided by NS2 and were modified to receive a transaction, response or update (in case of the children) in a package. These nodes have an agent and an application that are in charge of the network communication primitives (see class description below). The application code was modified to implement the communication module.

The main components of each server are shown by the Figure 4.7.

![Figure 4.7 Class diagram of the proposed system](image)

- **fullTcpAgent**: it is a class provided by the NS2 simulator, it differs from the other TCP agents because it supports bidirectional data transfer and connections may be established and closed (SYN/FIN packets are exchanged).
- **DBServerAgent**: it inherits from FullTcpAgent, so it has all the properties and methods but adapted to what the proposed system needed.

- **clientAgent**: this class inherits from FullTcpAgent, so it is responsible for the data transfer between client and server.

- **telnetApp**: Applications sit on top of transport agents in NS2. There are two basic types of applications: traffic generators and simulated applications. In this case, the simulated application is used and it is called TelnetApp. Application/Telnet objects generate packets choosing the inter-packet times from an exponential distribution.

- **DBServerApp**: It inherits from TelnetApp class, so it has all its methods and properties but adapted to the simulator needs. This class is in the server side. In this case the message received contains a transaction, so it has to be processed in that way. The sent message contains a response, an update or a transaction so it controls this message generation as well.

- **clientApp**: It inherits from the telnetApp class. This class generates the transaction from the client and uses the clientAgent to send it to the server.

- **lockManager**: It implements the necessary methods to control the transactions execution. And it is part of the Server class.

- **dataManager**: It implements the methods to simulate the database behavior.

- **DatabaseReplication**: It controls the database partition and replication technique applied.

- **Server**: It implements the classes lockManager and dataManager. It contains a timer that controls the moment when to update the children.

- **Client**: It contains the appClient object needed for the communication between one client and the server. It calculates the execution statistics.
Figure 4.8. Object interaction for Lazy Protocols

Figure 4.9. Object Interaction for Eager Protocols

Figure 4.10 Object interaction for Active Replication
These classes interact in different ways depending on the replication technique applied. Figure 4.8 shows the object interaction for the lazy protocols where the lazy behavior can be observed. The client sends a transaction through the `clientAPP` and `clientAgent` objects, and it is received by the `DBServerApp` object, which passes it to the `server` object. It requests the next step to the `replicationDatabase` object which checks the replication technique applied and replies to the server with the next thing to do. The transaction is sent to the `lockManager` object and it is executed using the `dataManager` object. After this, the response is sent to the `server` object which sends a message to the `multicast` object. This sends the transaction to the rest of replicas. Meanwhile, the `server` object replies to the client using the `DBServerApp` and `DBServerAgent` objects. The eager primary copy is shown in Figure 4.9, and the main difference with the previous one is that the `server` object waits for the response from all the replicas at level 0 to send the result to the client.

Figure 4.10 shows the object interaction for active replication. In this case, the `server` sends a message to the `multicast` object which sends using the total order broadcast primitive at the
beginning of the execution process. This technique is eager and because of this, the delegate server waits for the replicas to reply. In case of the certification technique (see Figure 4.11) the object interaction is different. A condition is added to the diagram. If the transaction commits, then it is sent to the replicas using total order broadcast; otherwise, the response is sent to the client. Weak voting technique (Figure 4.12) shows the different moments in which the server object asks the multicast object to send the total order broadcast. The first message is sent before the commit moment. The second message is sent with the result of the transaction.
Chapter 5

Evaluation

This chapter presents the performance evaluation of the proposed system using the simulator developed. The experimental settings will be described in section 5.1 and the results of the experiments will be explained in section 5.2 which were based on the replication technique used and the server load. The results evaluate the response time of the system, the abortion rate and the system throughput.

5.1 Experimental Settings

We performed an extensive set of simulations to compare the different replication techniques explained in chapter 2. All techniques shared the same infrastructure and the same parameters. The main performance metric is the response time observed by the clients.

Simulations were run with the parameters read from an XML configuration file. These parameters are listed and defined below.

- Total size: Database total size in GB.
- Copy size: Size of each database partition in GB.
- Replica number: Number of replicas of each partition.
- Protocol: Replication technique used for the current simulation.
- Client Number: Total clients in the simulation. It must be the same number of clients for every server.
- Write factor: Indicates the proportion of write operations within a transaction. If the value is 0 then is a read-only transaction.
- Write time: System time used for a write operation in ms.
- Read time: System time used for a read operation in ms.
- Hotspot objects: Number of operations per transaction that go to the hotspot.
- Simulation time: Total time for the simulation in seconds.
- Time between transactions: Time between the start of two consecutive transactions in ms.
- Operations: Number of operations in one transaction.
- Hotspot: The number of records in the hotspot.
- Freshness: Degree to which clients accept previous versions.
- Actualization interval: Time between server actualization to its children in ms.
- Maximum capacity: Maximum capacity of server processing.
- Minimum capacity: Minimum capacity of server processing.
- Level number: Levels in the hierarchy.
The specific parameters values used for the simulations are shown in Table 5-1. To calculate the number of partitions we took the “total size” parameter and we divided by the “copy size” parameter, in this case we had 25 partitions. Each one of the partitions has 4 replicas which is obtained by the “replica number” parameter; this means that we have 100 servers in the simulation. The database has a hotspot which contains the number of rows indicated by the “hotspot” parameter; each transaction needs to access the number of objects designated by the “hotspot object” parameter. Every delegate server has 6 clients which gave us a total of 600 clients. Each transaction generated by the clients has a random number between 10 and 20 operations (“operations” parameter). Each client waits an interval of 0.2 ms after it receives the server’s response to send the next transaction; this is indicated by the “time between transactions” parameter. The total time of a transaction execution is given by the “write time” and “read time” parameters along with the number of operations that are included in it. Each simulation has a write factor for all the transactions executed which varies in the interval presented in table 5.1 for the “write factor” parameter. If the value of “level number” parameter is 0 then is a simulation with no hierarchy. On the contrary, if its value is not 0 it means that a hierarchy should be implemented. If a server hierarchy is applied then, when a server reaches the “maximum capacity” means that a server from the next level has to be promoted. In case of a server reaches its “minimum capacity” means that a server for the next level should be shutdown. The servers in level 0 send an actualization to the servers in the next level every 0.4 ms which is given by the “actualization interval” parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total size</td>
<td>750 GB</td>
</tr>
<tr>
<td>Copy size</td>
<td>30 GB</td>
</tr>
<tr>
<td>Replica number</td>
<td>4</td>
</tr>
<tr>
<td>Protocol</td>
<td>1-6</td>
</tr>
<tr>
<td>Client number</td>
<td>600</td>
</tr>
<tr>
<td>Write factor</td>
<td>[0, 0.25, 0.5, 0.75]</td>
</tr>
<tr>
<td>Write time</td>
<td>2-12 ms</td>
</tr>
<tr>
<td>Read time</td>
<td>2-12 ms</td>
</tr>
<tr>
<td>Hotspot object</td>
<td>3</td>
</tr>
<tr>
<td>Simulation time</td>
<td>500-1000 seconds.</td>
</tr>
<tr>
<td>Time between transactions</td>
<td>0.2</td>
</tr>
<tr>
<td>Operations</td>
<td>20</td>
</tr>
<tr>
<td>Hotspot</td>
<td>2000</td>
</tr>
<tr>
<td>Freshness</td>
<td>[25, 50, 75, 100]</td>
</tr>
<tr>
<td>Actualization interval</td>
<td>0.4 ms</td>
</tr>
<tr>
<td>Maximum capacity</td>
<td>80%</td>
</tr>
<tr>
<td>Minimum capacity</td>
<td>20%</td>
</tr>
<tr>
<td>Level number</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5-1 System configuration
5.2 Results

In this section, we detail some results of applying the proposed replication techniques to a cloud system architecture.

The results show the response time, abortion rate and system throughput. The response time metric tells us how fast the system responds to the client, as faster is the response time, the better the system is. However, the ideal scenario is where all the transactions are successfully executed and we obtain this measured by the abortion rate. In general, the desirable system would have a short response time with a low abortion rate. Finally, the throughput tells us how the system can handle the load that it receives. Ideally, if we injected a load of X TPS in the system, we wait that the system works to X TPS; nevertheless, having the concurrency control and the replication technique causes that the system cannot handle the X TPS. We look for a system with the best throughput.

To evaluate the system behavior in regard to the previous three aspects, we varied the percent of update transactions; with this, we can estimate the effect of propagating the actualization to all replicas. To achieve an increase in performance applying the hierarchical architecture, we added a freshness factor that tells us the percentage of transactions that can be executed in different replication levels.

Varying the writing factor we have three scenarios: the first one is update intensive; the second is half of read-only transactions and the other half update transaction, and the third scenario consists in a read intensive load. This variation applies to both architectures with hierarchy and without it. In the case of having hierarchy, we varied another parameter: the freshness. This variation gave us the opportunity to make a correct evaluation of the performance gain applying the hierarchy.

5.2.1 Response Time results

Results without hierarchy

The first evaluations were made with no hierarchy. In this case, the experimental settings mentioned in the previous section were applied except for the level number, which was set to 0. Figure 5.1 shows the results for all the replication techniques explained in Chapter 2 executed with different write factor. As it can be seen as we incremented the write factor the response time was incremented as well. These results are similar to the obtained by Weismann and Schiper in [2]. As it was expected the active replication technique had the worst performance for the write factors applied. This is because the delegate server has to wait for the response from all its replicas to send the response to the client; besides, the read operations are also sent to the replicas, so they have to execute as well taking more time. On the other hand it is well known that primary copy does not have a good performance in a highly update scenario which is the case. The lazy has the best performance but there is a cost: the abortion rate; which will be shown and explained later.
However, the weak voting technique has a very good response time. This is caused by the fact that the delegate server does not have to wait for the replicas response. It executes the transaction and sends the response to the client having as response time the execution in the delegate server. With a certification technique, the certification phase along with the total order broadcast allows the servers to assure that every transaction will have the same result in all replicas. However, its time responses are higher than in a weak voting technique due to the processing required to certify that there is no conflict with another transaction and can commit, or in effect there is conflict and must abort.

Figure 5.1 Response time depending on the write factor parameter. With no hierarchy.

**Results with hierarchy**

With the same configuration parameters that were used for the non hierarchical simulations, we executed several simulations but this time we added hierarchy. The Figure 5.2 shows the different results for a write factor of 0.75 and different freshness parameters. As it was explained before, this parameter indicates the transaction percent that accept old values. The results show that for a high number of writes (0.75) and low freshness (25%) the response time is similar to that obtained with a non hierarchical architecture (Figure 5.1). This can be seen in Figure 5.2a. This is because the freshness is not high enough to balance the high number of update transactions with the readings executed in lower levels. To explain this, observe the Figure 5.2d where the freshness is 100% and there is a significant gain in the response time. This means that
the servers in level 0 only execute the update transactions, while the servers in lower levels execute the readings obtaining a gain in the response time. Now, if we compare Figure 5.2d with Figure 5.1c (both with write factor 0.75 but the first one without hierarchy and the second with hierarchy) we can see how the replication techniques behavior is similar, the worst is active replication and the best is lazy, but in the case of having hierarchy even active replication was around 30% better in response time.

As it was expected, when the write factor was decreased, the response times were decreased as well. Figure 5.3 shows the response time for the same replication techniques but decreasing the write factor to 0.5.
Figure 5.3 Response time depending on the write factor and freshness parameters. Having hierarchy with write factor=0.5

The Figure 5.4 shows the results for applying a write factor of 0.25. The performance obtained is above all, the best of all implementations. This is important because for a cloud system the majority of operations are read-only transactions. Even if we have a 25% of read only transactions accepting old values (Figure 5.4a) we obtain a significant gain in the performance comparing with no hierarchical architecture (Figure 5.1a) or comparing with a hierarchical architecture with higher number of update transactions.

We can observe that in a scenario with a low amount of update transactions (0.25) the different replication techniques have a similar behavior; this was expected because the transactions that take the major server processing time are those with write operations. To explain this we can observe Figure 5.4d which shows the response time for 100% of read-only transactions accepting old values, certification technique and weak voting are very similar. The certification technique does not spend so much time in the certification phase as it has so many read-only transactions executed in lower levels.
45

Having hierarchy with write factor=0.25

Having a hierarchical replication the Figure 5.4d represents the best scenario (low number of update transactions and 100% of the read-only transactions accepting old values) while Figure 5.2a the worst scenario (high number of update transactions and 25% of read-only transactions accepting old values).

5.2.2 Abortion rate results

Observing the figures in the previous section we could say that lazy techniques have the best performance. However, we have to look at its abortion rate (Figure 5.5) where it is the highest one. Therefore, lazy techniques are best suited for non-conflict scenarios. Nevertheless, we have a big number of concurrent and conflicting transactions. On the other hand, the weak voting replication has one of the best performances so far and the abortion rate showed in Figure 5.5 is the lowest even with high writing factors as can be seen in Figure 5.5c.
5.2.3 System throughput results

Finally, we evaluate the system throughput of all replication techniques in a hierarchical architecture with a write factor of 0.5 and 50% of the read-only transactions accepting old values (see Figure 5.6). Even when lazy has the best result, as it was explained before it has the highest abort rate, so we highlight the results for the weak voting and certification techniques which are the higher of the rest of replication techniques.
Summing up, we have confirmed that by having a hierarchical architecture with several levels we can achieve a better response time by mitigating the load that replicas participating in the replication technique in the first level have to afford. However, some replication techniques are affected by the increase of the number of replicas; this is the case of active replication. In this technique the delegate server has to wait for the responses of all its replicas. Moreover, as the replica number increases by promoting the replicas in lower levels, the response time is increased since all they have to do the same work.

The lazy replication technique has the best response time and the higher throughput but also the higher abortion rate. As it was explained before this technique applies only for non-conflict scenarios which is not the case. On the other hand, certification techniques have good results for this hierarchical architecture even if we have a high writing factor. These results were not as good as the weak voting techniques which have the best result in response times and in abortion rate. Of course, achieving an optimal performance would require an in-depth study of the system’s behavior under different scenarios, in order to provide the metadata manager with the necessary knowledge base to take the adequate decisions to adapt configuration parameters to the workload at each time.
Chapter 6

6 Conclusions and Future Work

6.1 Conclusions

We presented a study of transactional database replication techniques adapted to the elastic cloud environment. Based on this, we can conclude that there are some areas to explore, specially taking some classic techniques and combining them with new technologies to satisfy the current application demands.

We have applied the classic database replication techniques to a hierarchical architecture to simulate an elastic environment for a replicated/distributed database to provide a highly scalable and available service. Following the cloud paradigms of pay per use, the proposed system also features an elastic management of resources. This is accomplished by shutting down the database replicas that have not been used and turning them back on only when it is necessary to satisfy the client demands. Also, we have proposed a replication technique based on epidemic updates, which is able to provide different consistency levels according to the requirements of each application and, thus, build a replication hierarchy tree.

The experiments performed using the developed simulator have allowed us to verify that the existence of a hierarchical architecture working with asynchronous updates is able to alleviate the scalability limitations of traditional replicated database by redirecting transactions that tolerate less recent versions of data to the replicas in the next level.

6.2 Future work

The analysis and development of the replication techniques of a transactional database system applied to the cloud has to be continued in order to get a real implementation. In the following, we propose several directions for future research.

Test the architecture in a standard benchmark. In the evaluation of the transaction replication techniques applied in the cloud, a simulator has been used in order to point out the advantage of having a hierarchical architecture. However, the use of standard benchmarks, like TPC-C or YCSB, would be interesting in the presentation of the performance results.

Design a load balancer and a fully operational metadata manager. The metadata manager developed only implements a static partitioning and establishes the hierarchy at startup time. It should include the functionality of dynamically adjusting the partitioning scheme by splitting or merging data partitions. A load balancer can be added to the system which could help to gain in performance determining to which replica the transactions should be sent.
A hybrid system should provide a better use of resources. In cloud systems, the load for each partition is not necessarily the same. Because of this, a hybrid system could have a much better performance as it can apply a lazy technique for a partition which has a lower number of conflicts; while another partition with a higher load could apply weak voting to resolve the client requests.

Add a crash handler. In this study the system failures are not managed. It would be interesting to model the possible crash of the replicas and apply a recovery protocol to study the effect in the system performance.

Performance metrics. It would be interesting to apply other metrics to the system. For instance the freshness level in transactions is stated in terms of the hierarchy level that can manage each transaction. However, more complex formulations for data freshness could be applied, such as associating versions with timestamp [27].
References


[38] Chunye Gong, Jie Liu, Qiang Zhang, Haitao Chen and Zhenghu Gong. The Characteristics of Cloud Computing. 2010 39th International Conference on Parallel Processing Workshops.
