ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

INGENIERÍA INFORMÁTICA

Proyecto Fin de Carrera

LANDING TRANSACTIONAL SUPPORT ON THE CLOUD

"Desarrollo de un sistema de replicación de bases de datos en entornos dinámicos: particionado y protocolos de replicación asociados"

Alumna: Ainhoa Azqueta Alzúaz

Tutores: José Enrique Armendáriz Íñigo
Joan Navarro Martín

Pamplona, June 18, 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, the cloud platforms serve a huge variety of applications that need to store data in a structured way, mainly in databases. During the life of those applications, the amount of users accessing to date and the data size itself vary depending on several features: the novelty or the impact of applications or the seasoned nature of data (holiday booking, big event, etc.). Thus, The Database Manager Systems (DBMS) have to adjust the resources that each application has in order to minimize the amount of resources while ensuring a proper quality of service to the clients. The requirements of each application are independent so it is possible, though not easy, to balance or share resources.

Elasticity is the ability to deal with load variations in a live systems, by adding more servers during high load and the consolidation to fewer servers when the load decreases. This feature is so important in pay-per-use cloud infrastructures in order to minimize the operational cost. All this processes must have the less impact as possible in the given application as well as in the others that are running in the DBMS.

One of the main approaches followed was to replicate database in several physical nodes and coordinate the execution among them [34, 22, 26, 16]. However, the cost of propagating updates to all replicas and coordinate the execution of transactions [20] let to partitioning the data [31, 32, 2, 28]. This solution does not satisfy the demands of elasticity and scalability demanded by cloud applications. At this point, two alternatives emerges: the NoSQL data repositories (PNUTS [8], BigTable [7], Dynamo [17]) and the use of in-memory databases such as Google Megastore [3], Google Spanner [11], Cloud SQL Server [4], Elastars [15], Relational Cloud [13].

Two years ago the ”Grupo de sistemas distribuidos” research group, at Universidad Pública de Navarra, started with the development of a database and a system that allow obtain transaction support to the aforementioned applications, it was called ”Urraca”. In this work, we have improve some parts of this initial project resulting in a new project, Zoo. It is split into three main components: the client tier responsible for executing transactions on the client side; the Metadata Manager, which is the core tier, responsible for maintaining the systems health and response to the clients the location of the data for each of their transaction; and, the replica tier in charge of executing the transactions and maintain data consistency.

1.1 Motivation and Goals

We took the Master thesis of I. Arrieta-Salinas and M. Louis Rodríguez as a starting point for this project. We are going to deploy a distributed database to be used in a cloud environment as a specific case of Platform-as-a-service.

We assume that data is partitioned and several replicas store a copy of a given partition. The clients issue transactions by means of a standard library such as JDBC. To do so, they need information about the data placement that is managed by a Metadata Manager. The Metadata Manager manages the partitioning and the replica placement among all replicas building a replica cluster on each partition. The replication cluster has a few replicas running a replication protocol to provide strong consistency and the rest receive the propagation of updates in a lazy manner. These replicas are logically constituted as onion layers around the core replicas running a given replication protocol.

The implementation of this system had several drawbacks that we try to fix in this work. First of all, clients an the MM need to be physically in the same machine which leads to a penalty performance in heavily loaded scenarios. The system was optimized for YCSB [9] that consisted in transactions with a single operation and they are run over two replication protocols: primary copy and active replication [34] that are known to perform badly update intensive scenarios. Moreover, there was no load balancing at all according to replica performance, it was merely a round-robin policy among all replicas at the core level.

We try to argument the system limitations (described in more detail in Section 2.1) and to going into the system implementation. This is going to be explained in the rest of this work.

The main goals of this project are focused in the different parts of the system. In regard to the Client Module, originally the client was the OLPT-Benchmark, a module that consist in send specific types of transactions to the system by a JDBC connection. In the actual version this module has been modified allowing to the transaction to have more than one operation and several parameters has been introduced to the transaction which allow to the system to treat them differently. Respecting to the Metadata Manager one of the main goals between the others developed in this project is the decentralization of the Client and Metadata Manager modules physically. The rest of modifications are the creation of a structure that allow to the Metadata Manager to know the architecture of the Replicas Cluster and the development of a new ReplicaChooser function based on the CPU charge allowing a correct load balancing. And finally in the Replicas Cluster has been implemented new protocols that have permitted to run different replication protocols in different partitions simultaneously without the knowledge of the Client and the Metadata Manager.

1.2 Contribution

Thanks to the project the different replication protocols that appeared in the literature will be associated with a certain replication database structure obtaining the better throughput as possible.

In the academic world this project could be used in distributed systems subjects in order to analyze how the different replication protocols works and to understand the distributed databases and the different properties that should obey in order to maintain data consistency, reliability and availability.

1.3 Document Organization

The rest of this document is structured as follows. Chapter 2 is going to show a sum up of the state of the art in the field of distributed databases. Chapter 3 introduces the system structure and the main concepts in order to follow with the project development. Chapter 4 describes all the new implementations that have allow to achieve the main goal of this project. Chapter 5 is devoted to explain and discuss the main results obtained from our experimental setup. Finally, conclusions and future lines are presented in Chapter 6.

Chapter 2

Related Work

We are going to review the current state of the art in distributed storage systems in the cloud in this chapter. In Section 2.1, some of the current transactional support approaches are discussed in order to portrait the current situation in this field. Section 2.2, describes current key-value stores in the cloud aimed to provide high scalability though they do not offer transaction support.

2.1 Current Transactional Support in the Cloud

The necessity of having scalable data management systems has yielded to the apparition of new systems able to manage large amount of data such as: BigTable [7], PNUTS [8] or Dynamo [17] just to mention a few. They only support key-based access and ensure high availability and high scalability but lack of transactional support.

More recently there are still many applications that do not fit in these systems. More precisely, they cannot so easily resign from their transactional behavior. Thus, researchers and industry have studied very first approaches to support transactions in the cloud and appeared: Google Megastore [3], ecStore [33], Elastras [15], Relational cloud [14]. Google Megastore provides transactional multi-key accesses using BigTable as base of the system, uses a Paxos [24] protocol for synchronous replication of each write in the system. It allows seamless fail-over between nodes but it offers higher transactions latencies. ecStore is an elastic cloud storage system that supports automated data partitioning and replication while supporting transactional access across rows. Elastras, is an elastic database system capable of scaling up and down according to the workload, the kind of transactions supported are those that access a single partition, although they provide a similar mechanism to Sinfonia [1] to execute transactions accessing several partition (called mini-transactions).

2.2 Key-Value Stores

Key-value stores are based on key-value data model and single key access guarantees, and are able to achieve high performance and availability. These systems relax ACID properties

(see Section 3.4.1) and rely on eventual consistency. The main goal of this systems is to ensure data availability before ensure strong consistency.

Some of the Systems that can be found are Amazon's Dynamo [17], the only kind of operations that supports are read and writes over key-value records. Google's Bigtable [7] that achieve high performance and availability by giving up some functionalities such as joins and ACID transactions. PNUTS [8], developed by Yahoo!, that has a feature that allows to the rest of replicas to receive the messages in the same order as the master has managed. This system also provides to clients a selection of the freshness levels while issuing read operation. Recently, there has been developed the open source counterparts to this industry reposition such as HBase [21] and Cassandra [23].

Chapter 3

System Model

In this chapter the main parts of the project are going to be explained one by one. In Section 3.1 the motivation is going to be explained, whereas Section 3.2 presents the system architecture. Sections **??** and 3.6 are going to show the interactions between its internal modules, the Replication protocols, the CAP Theorem and detail how partitions are stored in main memory.

3.1 Motivation

As will be explained in Section 3.4, different replication protocols in the literature are better or worse in terms of efficiency, depending on the load of the system; i.e., the primary copy replication technique has better performance in scenarios with read intensive workloads. As opposed to scenarios with high rates of update transactions, in scenarios that receive a load with a great number of write transactions, are more convenient the replication techniques based on group communication primitives.

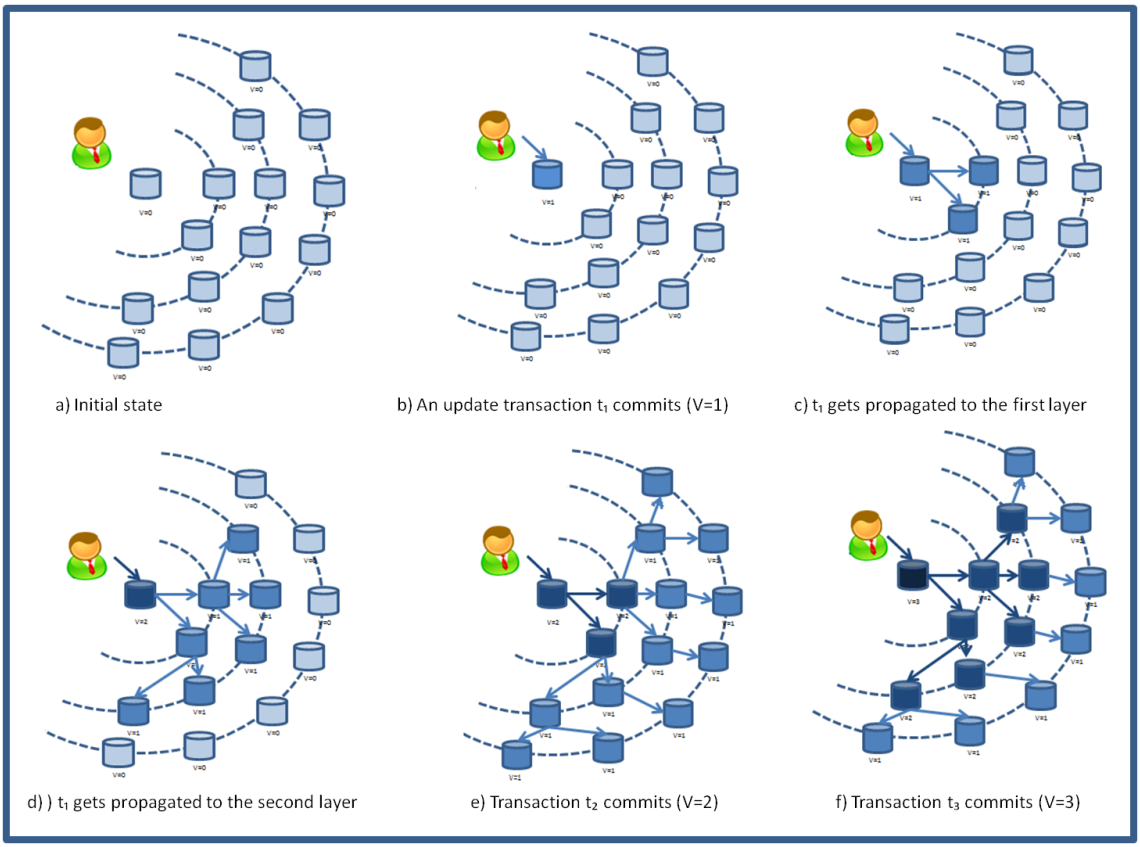
However, the aforementioned protocols have serious problems of scalability. The main factor is the cost of propagating updates to all replicas. This cost is proportional to the number of replicas; in particular, update everywhere replication protocol needs several rounds of messages to reach consensus in the order in which messages are delivered. This can be alleviated by allowing a set of all replicas to behave as primaries so that the number of replicas that must agree on the order of the messages are less than in the original case. The others will act as backups replicas, so that propagation techniques updates made in primary replicas are relayed to rest.

Consider a system as shown in Figure 3.1, in which there are 4 levels of replicas. Each of these levels will have a different version of the database as time goes by. Let see the a) state corresponding to the initial state of the system, all levels are at version 0 of the database. Now suppose that a customer wants to make a transaction *t1*, this transaction modifies the replicas that belongs to the level 0 or core level (b) state). After a certain time the update that modified the core level replicas is propagated to the replicas that are at the level immediately below, modifying the database from version 0 to version 1(c) state). As the system progresses, the propagation of the transaction *t1* continuous its course by changing the state

of the database (d) state). Then, the client starts a new transaction *t2* updating the replicas that are in the core level of the system, the propagation of *t1* follows its natural course by changing the state of the database until it reaches the last level of the system (e) state). Finally, in f) state we can see that the client starts a new transaction *t3* and as *t2* continuous with propagation through the different levels.

This system is still being developed and implemented. It tries to get better results in performance, offering greater data availability and fault tolerance system to the customer, and all this in a completely transparent way to the client. The client connects to a middle-ware that indicates int which partition the transaction must be executed. All replicas use a JDBC connection to a PostgreSQL database so the customer has the feeling that he is connecting directly to the database. Moreover, in the rest of the chapter we are going to describe each component in the system.

Figure 3.1: Updates propagation example



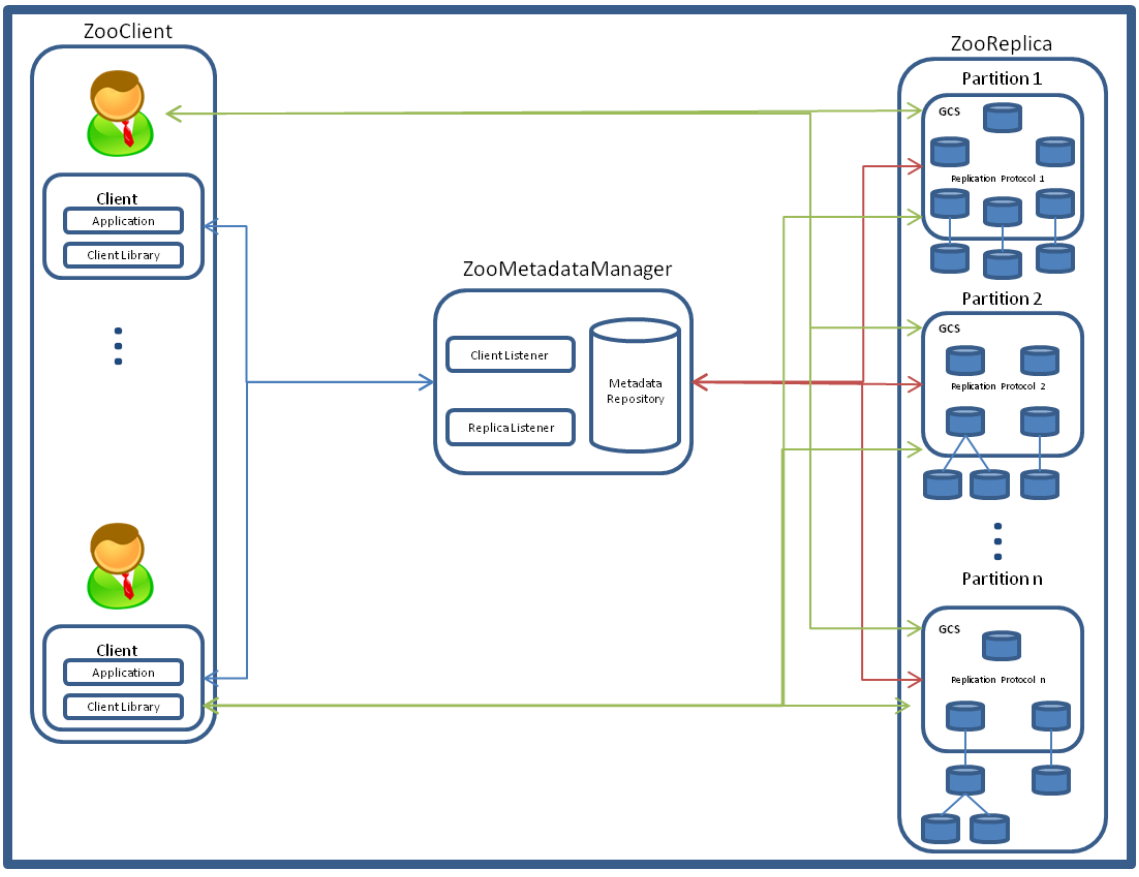
This system is composed of three parts or modules that later in Section 3.2 are going to be specified in more detail. But with the aim of providing an overview, the different modules are described. The first one is the ZooClient module, each client uses a JDBC connection with the driver that is going to be communicated with a middle-ware called ZooMetadataManager. This last module is in charge of attending all the request that clients send indicating in with

partition the transaction must be processed. And the last but not least, the ZooReplica, is in charge of executing the transactions that clients send.

3.2 System Architecture

This Section introduces all the components of the system. As it can be seen Figure 3.2 there are three parts that are clearly differentiated. The *Client Module* ZooClient, that is the responsible for simulating the behavior of various clients launching transactions to the system. The *Metadata Manager Module*, called ZooMetadaManager, who is the coordinator of the system. And the *Replica Module* ZooReplica, that will allow us to perform transactions in different partitions in which will divide the database.

Figure 3.2: System Model



3.2.1 Client Module

The Client Module is the engine of the system. It is in charge of simulating the behavior of multiple clients by sending a large number of requests per second to the system in order to

submit it to various stress levels. This module is mainly divided into two parts:

- **Client part:** This part of the module is in charge of reading from composed of transactions that the system will have to run. Once the transactions are stored, a connection is established with the JDBC driver, in our case ZooDriver, but it can be executed directly on the PostgreSQL driver. Then, for each of the clients that has been configured by reading the file, transactions are sent through the driver waiting for an answer. All information regarding with the response time, number of aborted transactions, etc. is stored in a file that will later be processed by another module of the system to obtain a comprehensive analysis of the system performance.

- **Driver part:** This section of the module is responsible for sending the transaction that has been received from the ZooClient to the Metadata Manager Module in order to obtain the necessary information to run the transaction in the most convenient replica, this process is explained in Section 3.2.2. Once this process is over, it sends the transaction to the selected replica and waits until its outcome, sending to the ZooClient the result.

3.2.2 Metadata Manager Module

The Metadata Manager Module (MMM) is the brain of the system. It is in charge of specifying to the client where the transaction should be executed, to know the structure of the system and to know the state of all components. All these features are explained with more detail in the paragraphs below:

- **System State Storage**

  This component allows the MMM to know how many replicas are alive, in order to inform the client the best suited replica for its request. This process consists in the reception of heartbeat messages that the replicas send to the MM, containing the replicaID, the CPU charge, the average of transactions executed per second, the average of freshness and the average of read only transactions.

- **Mapping of partitions**

  This function can be separated into two different specifications. The first one is necessary to answer to the client with the replicas in which the transaction is going to be executed, this process is going to be explained in Section 4.4. And the second one is a mapping of the different partitions and the replicas that belong to each partition, this information is obtained by the MM configuration file.

- **Server client request**

When a client sends a transaction, it firstly arrives to the MMM. The MMM has to answer to the client with the information of the replicas in which the transaction must be executed. To carry out this test the MM analyzes the different operations that belongs to this transaction and looks through the PDT structure, that is going to be explained later in Section 4.4, in order to find the partition that contains the register or registers accessed by the operation. Once these partitions have been found, a process, called ReplicaChooser, chooses the best replicas and this information is sent back to the client.

3.2.3 Replica Module

The part of the system that is in charge of executing the transactions received from the client is the Replica Module. When a transaction is received from the client, the transaction is processed as follows:

1. It is tested if the transaction is a read-only transaction or not.

   If it is a read-only transaction, it is executed via the JDBC and the result is sent to the client.

   If it is an update transaction, it is processed by the replication protocol that this replica is running in the case that the replica belongs to the core level.

2. Once the transaction has been executed the result is sent to the client, and simultaneously it is sent to the rest of replicas that belong to the core level in the case the transaction is not-read-only. When this process has finalized the write-set of operations is propagated to the rest of levels.

The client has the impression that it is executing the transactions in a centralized database and the not-read-only transactions are distributed in all the partition so the system is always updated.

Each replica that belongs to the core layer of its own partition have running a given replication protocol (the family of replication protocols will be explained in Section 3.4). Hence, when an update transaction arrives to the replica the write set is sent to the rest of replicas in the core layer.

3.2.4 Updates Propagation

Once the core level replica has executed the updates, the write-set is transferred to its own children replicas by sending messages through the point-to-point connections, in the hierarchy in order to propagate the updates. The set of incoming write-sets are stored until a

given threshold is reached, and the write-sets are propagated and executed in the secondaries replicas in the same order.

3.3 Communication between the Modules

All those modules must be connected, due to the next cases:

1. Client-MM Connection: The client process sends request messages to the MM process in order to know where the transaction should be processed.

2. Client-Replica Connection: The client process sends transaction messages to the different replicas that should execute the transactions.

3. Replica-MM Connection: Each replica sends status messages to the MM in order to inform that they are alive andand share other useful information such as CPU.

4. Replicas Connection: All replicas should be connected with the rest of replicas in their partition in order to broadcast the read-set message to the rest of replicas in the core level and to propagate the updates, see Section 3.2.4.

3.3.1 Point-to-Point Communication System

All different components of our system should be in contact in order to share information by exchanging messages. The behavior is very simple: each module has one or more ports waiting to receive messages from the rest of modules. When one module wants to send a message a socket is opened with the received port and the message is delivered.

The procedure is as follows: an asynchronous FIFO quasi-reliable point-to-point channel is assumed. There are two functions that allow the communication: PTPConnectionSend(m) and PTPConnectionDeliver(m), where the first one allows to send the message to the receiver and the other one allows to receive the message from the sender part.

Two kind of connections are going to be differentiated in this project, 1) when then sender is waiting for an answer, so the connection is not closed till it receives the answer and 2) when the sender has sent the message, so de connection is closed.

3.3.2 Group Communication System

One of the main problems that we have in our system is to find a way to send to the level core replicas, the messages in the same order. And this problem has appeared due to the fact that

we need that all replicas must execute all the transactions in the same order. This problem cannot be solved with a PTP Communication System as we have seen in Section 3.3.1, and that is because we need a consensus process to decide the right order in with the messages should be held in the core level replicas.

The solution is to use Group Communication Systems (GCSs), a set of tools that allow us to send messages between multiple members in the same group of distributed process. To better understand why the GCSs have been selected to solve this problem, some of their main characteristics are detailed below:

1. **Communication Service**

   The *Communication Service* gives us a set of multicast protocols to send the messages between the different members of the group. In the literature [18] appeared three different reliability degrees to deal with the delivery of multicast messages:

   - **Unreliable multicast:** It does not prevent message losses or drops. It is the lowest degree of reliability.

   - **Reliable multicast:** This reliability degree claims that if a message has been multicast in the group, all members are going to receive the message. It has three basic properties:

     (a) Validity: If a correct process multicast a message m, then it eventually delivers m.

     (b) Agreement: If a correct process delivers a message m in view v, then all correct members of v eventually deliver m.

     (c) Integrity: For any message m, every correct process delivers m at most once, and only if m was previously multicast by a process.

   - **Uniform reliable multicast:** It provides the strongest degree. It claims that if a message has been multicast by a member of the group and it fails, the rest of members in the group, that are available, are going to receive the message.
     This process needs another action, and it is that the GCS has to know that all alive members in the group have received the message, which means that there is going to be more latency in the system due to the high quantity of messages that should be interchange. Two properties of reliable multicast are required by it:

     (a) Uniform Agreement: If a process delivers a message m in view v, then all correct members of v, eventually deliver m.

(b) Uniform Integrity: For any message m, every process delivers m at most once, and only if m was previously multicast by a process.

Having a general point of view of the different reliability degrees we have to say that the best is the last one, due to that it ensures that no false updated can be executed in the system.

As it has been explained previously, all replicas must receive the messages in the same order, so there are some ordering guarantees that constraints the order in which the GCS delivers messages:

- **FIFO Order:** First In First Out Order. The messages are delivered to the members in the same order that it has arrived to the GCS.
- **Casual Order:** This is an extension of the FIFO order, the difference is that when a member has received one message, a response message is sent to the GCS in order to notify that he has received the message.
- **Total Order:** Guarantees that all members are going to be delivered the messages in the same order, irrespective of which one has multicast them.
  A combination with the two previously ones can be created: FIFO Total Order and Casual Total Order respectively.

2. **Membership Service**

   The aim of the service is to create a group or view with processes that want to be part of it. The group can change throughout all the process, it increase its number of members when a new process logs in or decrease when a process leaves or crashes. The GCSs are view-oriented, which means that when a change occurs in the group they are notified sending view changes messages. So we can define the view as the representation of all group membership.

   There are two different kinds of membership services, in terms of group composition:

   - **Primary partition services:** All the processes in the group have always the same view due to the fact all processes in the system are totally ordered.
   - **Partitionable services:** The processes in the system are partially ordered, so when multiple disjoints occurs the processes have different views of the membership.

3. **Virtual Synchrony**

   This property claims that if there is a view where a new process wants to join to it and a message needs to be multicasted. The GCS waits till the new process has been joined to the view correctly. More formally:

   If two processes p and q install the same view v in the same previous view v, then any message receive by p in v' is also received by q in v'.

   So, view changes are seen as synchronization points, in the set that multicast messages are ordered with respect to view changes. Process that installs view v in view v' and has received a set of multicast messages M between those two view knows that all processes that are in both v' and v also receive M between those changes.

3.4 Replication Protocols

As stated above, the replicas that are in the core level will be running replication protocols, this is why we are going to have a look at some of the protocols that can be found in the literature [34].

Each of the replicas must receive requests in the same order to ensure consistency in the database, one way of achieving this is using a group communication primitive called Total Order Broadcast (TOB) which ensures that messages are received by all replicas in the same order. Therefore, all protocols that we study below use such primitive, see Section 3.3.2.

Replication protocols based on TOB that are going to be detailed are: Active Replication, Certification based and Weak Voting.

- **Active Replication:** In this technique the client *c* sends the transaction *t* to the delegate server *sd*, when it receives *t* it is broadcast to the other servers by TOB. When these have received it, they process it and the *sd* sends the result to the client. As all servers receive all transactions in the same order if one transaction aborts the rest will too.

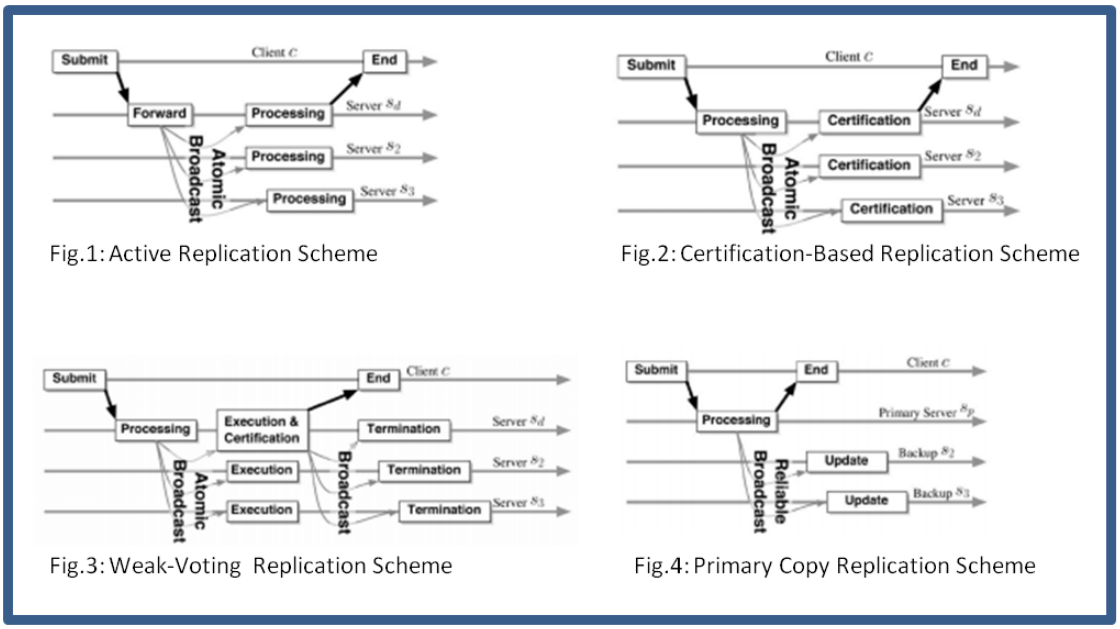
- **Certification-based Replication:** In this technique when the delegate server *sd* receives a transaction *t* from customer *c*, *sd* executes transaction's read operations, delaying the write ones and sends this write-set (*ws*) by TOB to the rest of servers. They certify if the transaction can be executed or not, if so each server execute the *ws*. Sd server sends to the client the outcome of the transaction.

- **Weak Voting Replication:** In this technique when the delegate server receives a transaction from the client, like the Certification-based Replication, executes the read operations and delays the write ones. Followed, the *sd* sends the *ws* to the other servers

by TOB and determines whether those operations can be executed and resend the information regarding to this decision (commit or abort) by TOB . When servers receive this second message they run the *ws* or not depending on the message's content. And the proxy server sends the client the result of the transaction. This technique is very expensive because it causes a lot of network traffic due to the double message that the *sd* sends to the other servers.

The following replication protocol presented is called **Primary Copy**, it is a traditional database replication technique. This one do not use a group communication primitive as it is used previously. The behavior is as follows, only one of the servers is elected as a delegate, it is called *primary server* and its unique responsible for processing transactions. The rest of servers are called *backups* and are responsible for receiving the updates. In this technique, there is only one server capable of absorbing all the transactions and may potentially become the bottleneck of the system and a single point of failure.

Figure 3.3: Replication Techniques Schemes [34]



3.4.1 ACID Properties in Distributed Databases

Database management systems (DBMS) must guarantee the ACID properties (i.e., atomicity, consistency, isolation and durability) for every database transaction.

The properties consist of the following: *Atomicity*, ensures that a transaction is executed or not, i.e.; if any of the operations that takes part of the transaction has not been executed

correctly, the entire transaction as a whole is aborted leaving the database in the same initial state. *Consistency*, this property ensures that a transaction starts in a valid state of the database and terminates its execution in a new valid state. *Isolation*, ensured that one transaction can not affect to others. This ensures that the execution of two transactions in the same information are going to be independent and without producing any kind of error. *Durability*, this property ensures that once a transactions has been completed successfully, the changes that have occurred in the database will not fall apart under any circumstances.

3.5 Consistency Vs. Availability: the CAP Theorem

The CAP theorem [5] states that consistency, availability, and partition tolerance are systematic requirements of designing and deployment of applications in a distributed environment. It also states that a scalable system can fulll at most two of these three properties. In the context of CDMSs, the CAP theorem results in the following tradeoffs:

- **Consistency & Availability:** When a CDMS is optimized for consistency and availability, this means that no requests will work on partial data. In the case of network or partition failure, requests will wait until partitions heal. In this case, the system latency is increased.

- **Consistency & Partition tolerance:** When a CDMS is optimized for consistency and partition tolerance, this means that some requests will work on partial data. Some requests will be refused. In this case, the latency of the system is reduced but its availability as well.

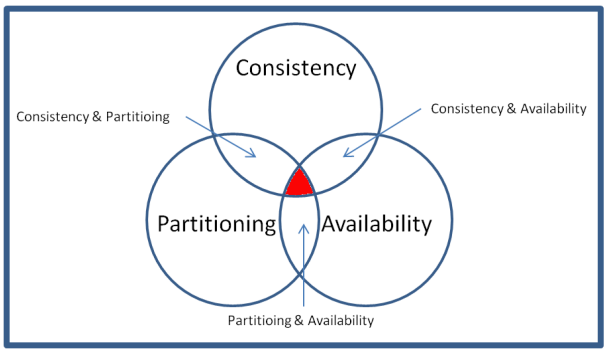
- **Availability & partition tolerance:** When a CDMS is optimized for availability and partition tolerance, this means that all requests will be answered in all cases. Requests may return inaccurate or stale version of data. In this case, the latency of the system is reduced and its availability is higher.

3.6 Storing Partitions in Main Memory

In some cloud solutions, replicas maintain the partitions in main memory to avoid writing disk, having to ensure durability. Other systems that rely on data structures stored in memory reduce the response time but periodically dump the data to disk.

Approaches in main memory storage allow the use of key-value storage and SQL databases. Are both possible solutions to provide transactional behavior when on replicated storage systems.

Figure 3.4: The CAP Theorem Ilustration



Chapter 4

System Specification

Once already described the system structure in the previous chapter, this one is focused on the detailed specification of the different components of the system. This project was initiated by Itizar Arrieta in his final master project [30] and Mariela Louis [29]. During this chapter an example is going to be proposed in Section 4.1, it is going to guide the explanation of the various changes that the system has undergone throughout the development of this project. What was intended was to make a much more configurable system that allows us to perform a variety of tests now and in future phases of this project.

4.1 A Database Application Sample

For the sake of clarity in the explanation of the different developed components, we are going to introduce an example of an application that will be used through the chapter.

We will have a database with a single key-value structure table, called *simplebm*, which will contain 30000 records. Each register is composed by an integer identifier as a key and character field of 1024 bytes as a value which results in a database of 32.4 MB. This will be divided into three equal partitions of 10.8 MB.

With regard to the generated workload. It has been generated a simulated workload of 800000 transactions where each transaction consist in two operations. We distinguish 3 different types of transactions:

- *Search:* This kind of transaction is composed by 2 punctual select operation, so it is a read-only transaction.

- *Check & Correct:* This type of transaction consist of 2 operations, the first one is a select operation and the second one is an update operation.

- *Information Exchange:* Two update operations are the ones that take part in this transaction.

4.2 The new ZooClient

The initial version of this project used the OLPTBenchmark [12] to simulate the workload application by way of different standards widely accepted in the industry and research community. It generates a series of transactions where we can tune several parameters in, such us: number of clients, number of transactions per client, and the time period in seconds that this tool has to generate.

We have changed the workload because it was clear for [30], [29] that the proposed architecture offered a join in terms of performance with the OLPTBenchmark. However, we want to emphasize that this should work even better in update intensive scenarios with clients requesting strict consistency. This was not possible to model with OLPTBenchmark, so we decided to implement a new client benchmark.

The new version works differently, when the client is launched a configuration file is read, see Figure 4.1. The clientManager process gets the following information: Driver connection configuration, the configuration of the several works that each client has,the name of the file in which transactions are found and the file name in which is going to be stored all the information obtained from the experiments that will be analyzed later.

Once the process has finished to access the configuration file, it stores the transaction workload in a structure that classifies to each customer its own transactions. Afterwards, the connection to the driver is created and launches for each customer a thread that is going to be responsible for running each transaction and storing the necessary information to obtain the statistics used to analyze the result of the experiments.

Figure 4.1: Client Configuration File

```
<configuration>
    <Connection
        driver="es.unavarra.gsd.zoo.driver.ZooDriver"
        url="jdbc:postgres://localhost/jungleDB"
        username="myuser"
        password="myuser"
        schema="public"/>
    <works>
        <work time="20" rate="50"/>
        <work time="300" rate="50"/>
        <work time="300" rate="50"/>
        <work time="300" rate="50"/>
    </works>
    <workload nameFile="queries_experiment.sql"/>
    <statistic nameFile="executionInfo_1_50_1.txt"/>
</configuration>
```

4.3 Metadata Manager Module (MMM) and Client Module (CM) Separation

At the beginning of the project both the MMM and the CM were together, in the same computer, due to this fact computer's cache memory was easily overloaded when the systems was running. So one of the main goals was to separate both modules.

To achieve this, an intensive study was carried out to define how both modules must communicate and define a communication interface so that clients and the MM does not need to be in the same machine. Each client can open a point to point connection, as explained in Section 3.3.1, with the MMM that is listening on a given port. When the MMM receives a request, it keeps the communication open until it computes the set of replicas where the given transaction must be executed and sends this information back to the client.

4.4 Partitioning the Distribution Table

As discussed in the specification, the project the database is going be stored in multiple partitions. Once the MMM receives a client transaction, it has to check in which partition the client must execute the transaction. This partitioning process can be done in "a priory" basis either: manually with offline artificial intelligence techniques; or, by another deterministic mechanism (locks, round-robin, etc.). The master thesis of Mariela Louis [29] developed these techniques alone with a set of data structures and functions to determine the partitioning information.

Let us see how the partitioning information is built with our example. At the beginning the MMM reads a partitioning configuration file (see Figure 4.2): Each entry of the file contains the partition identifier, the set of tables and the set of rows of each table that is stored. It shows that there are three partitions with 10000 records of the *simplebm* table, each one storing in ascending order, with a range of 10000, the whole database.

Concurrently to the file configuration processing, the MMM builds a data structure, called Partitioning Distribution Table (PDT), intended to quickly fetch the partitions (or partitions) where a client transaction should be executed and its composition is shown in Figure 4.3.

4.5 Replica Chooser Function

This is a MMM feature that chooses one replica among all that can execute a given transaction. Up to now, the replica was randomly chosen [30]. We have incorporated another functions that takes into account CPU usage of all replicas to determine the best one. Recall that the MMM collects this information by way of the "heartbeat" messages, received from the replicas. We humbly think this is an improvement of what it was already developed.

Figure 4.2: Partitions Distribution File

```
<partitions>
    <partition id="1" size="10000">
        <partitionConfiguration id="1" table="simplebm" min="1" max="10000"/>
    </partition>
    <partition id="2" size="10000">
        <partitionConfiguration id="2" table="simplebm" min="10001" max="20000"/>
    </partition>
    <partition id="3" size="10000">
        <partitionConfiguration id="3" table="simplebm" min="20001" max="30000"/>
    </partition>
</partitions>
```

Figure 4.3: Partitions Distribution Table

Simplebm	Min : 1 Max : 10000 Partition: 1
	Min : 10001 Max : 20000 Partition: 2
	Min : 20001 Max : 30000 Partition: 3

4.6 Allowing Several Operations per Transaction

The older implementation was best suited for the YCSB benchmark [10] where each transaction was composed by a single operation. This behavior does not satisfy our current workload needs and we have modified the system to tolerate several operations per transaction. Actually, the MMM gives the set of replicas where each operation can be executed. Thus, each operation is sent to the proper replica; although each replica checks if the given operation can be executed or not.

4.7 New replication protocols implemented

So far only protocols were implemented in the system: primary copy and active replication 3.4. We wanted to emphasize that several traditional database replication techniques can increase the performance of cloud-based databases. To this end we have implemented two additional replication protocols, both of them has been implemented following [34].

4.7.1 Certification Based Replication Protocol

The certification-based replication protocol developed is shown in Figure 3.3. Following with the method that Itziar Arrieta determined in her Master Project [30] for all the protocols implemented in it. The *processUpdTransaction(TransactionMessage tm)* method is in charge of execute the transaction without commit and to broadcast the transaction to the rest of replicas in the core level using TOB. The *processRegularMessage(RegularGCSMessage msg)* method works as follows: when the message is received each replica execute the certification phase, in case of commit the transaction is executed and committed in the replicas and the delegate one sends the result to the client. In case of abort only the delegate replica executes a rollback and sends the result to the client.

4.7.2 Weak-Voting Replication Protocol

The weak-voting replication protocol developed is shown in Figure 3.3. Where we have the main methods of this protocols and one more: The *processUpdTransaction(TransactionMessage tm)* method is in charge of execute the transaction without commit and to broadcast the transaction to the rest of replicas in the core level using TOB. The *processRegularMessage(RegularGCSMessage msg)* method works in two different ways if the replica is the delegate or not. In the case of the delegate it is waiting to receive the notification from the rest of replicas that have could execute the transaction in the affirmative case a message is sent to the replicas indicating that they have to commit and abort in the other case, and sends the result to the client. In the case of no delegate replicas, each one certify if there is any conflict between the new transaction and the other that have been executed, and executed the transaction sending a message to the delegate indicating if the transaction has been executed or not. The new method implemented for this replication protocol, is the *termination(ITransaction, String status)* method, it check the information received and execute the commit in case it has been indicated or a rollback in the other case.

Chapter 5

Experiments

The background of the chapter is organized as follows: Section 5.1 presents the system configuration details. The next section describes how we have established the different scenarios to be tested such as: the number of replicas per partition; the replication protocol running on each partition; or the replication hierarchy depth. Finally, Section 5.2.1 discuses the results obtained by the previous configurations with an analysis of the throughput, transactions response time and abortion rate.

5.1 System Implementation Details

Our testing configuration consists of 10 computers connected in a 100 Mbps switched LAN, where 6 replicas are guested in 6 computer that are equipped with an Intel Core 2 Duo processor at 2.13 GHz, 2 GB of RAM and 250 GB hard disk. Other 2 computers are equipped with an Intel Core 2 Quad processor at 2.66GHz and 3,5 GB of RAM. And the last 2 ones are equipped with an Intel Core i5 processor at 3.336GHz and 3,6GB, in which there are 4 virtualized machines with Virtual Box, each one have 1 core and 1GB RAM. These 8 virtual machines 8 replicas are going to be run. All replicas run the Linux distribution Debian 7.0. Each machine runs a Java Virtual Machine 1.6.0_45 executing the application code.

So the ips are distributed as follows, from 1.1.1.101 to 1.1.1.106 are the first 6 computers. From 1.1.1.107 to 1.1.1.114 are the 8 virtualized machines and 1.1.1.34 and 1.1.1.38 are the second ones.

5.2 Experimental Setup

The main goal of this project is to check the gain we obtain in the system with different possible configurations that can be made. As will be shown in Section 5.2.1, all experiments will have the same structure in the system, which will allow us to compare different configurations of protocols that will be running on the replicas that are in the Core level. In Section 5.2.2 is going to be explained the system load, i.e.; the configuration that will have the different transactions that will launch every client, but we have to take into account the two parameters that we will create the workload file: the percentage of update transactions

and the level of freshness. In terms of the ratio of update transactions, it indicates how many transactions of the generated script will be of update, this is going to assume that such transactions will only be able to run in replicas that are in the core level. The second parameter corresponding to the freshness level indicates what level transaction needs to be executed and obviously as explained for a transaction can be executed at another other than the core, it has to be a read transaction and the freshness level must be greater than 0, such that if the level of freshness is 1, the transaction will be executed in layer 1 and so on.

With everything explained above, the following parameters are going to be tested in each of the experiments:

- *Transactions per second collected:* This will allow us to see the total number of transactions in each of the experiments.

- *Response time:* This parameter will allow us to see the average response time of each transaction to be used in each of the experiments.

- *Aborts rate:* This parameter indicates the average number of aborted transactions we will have in the various experiments, we have to keep in mind that we want to serve the largest number of transactions successfully.

5.2.1 System Structure

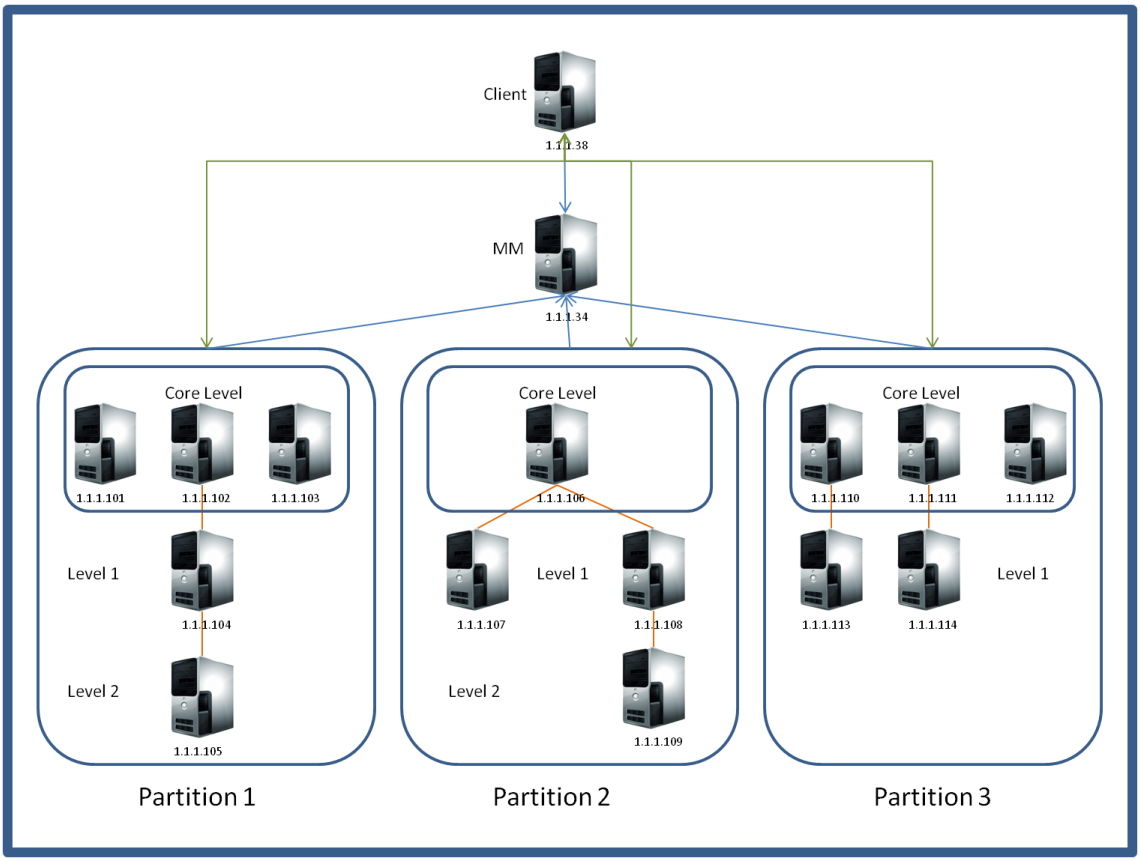
The system structure considered in all the experiments is shown in Figure 5.1. One machine is going to simulate the client behavior as it has been explained in 3.2.1, another machine is going to run the MMM and the rest of machines are going to be distributed as follows:

As we could see there are going to be 3 partitions with 14 machines in total. First partition have 3 levels, in the core level there are three machines were a determinate protocol is going to be running, in the second level there is one machine that depends on one of the replicas in the core level and in the third level there is only one replica. Second partition is distributed in three levels, but in this case only one replica is in the core level, two in the second one and one in the third level. Finally, in the last partition there are two depth levels where in there core one there are three machines and in the second one there are two machines whose antecedents are different machines of the immediately superior level (Figure 5.1).

5.2.2 Experiments' workload

Two different workloads are going to be tested in order to obtain the result for the best configuration and the worst:

Figure 5.1: System Structure



1. **Worst Case:** We are going to start with the worst case scenario where 80% of transactions require strict consistency and 80% are update transactions. This setting provides a software case since replicas located outside the core level are most of the time idle; i.e; we obtain no gain with the replication hierarchy.

2. **Best Case:** It this case the are only 20% of update transactions and 20% of the global workload are going to be executed in the core level. With this load we intent to alleviate the core level and earn in performance with respect of the worst case.

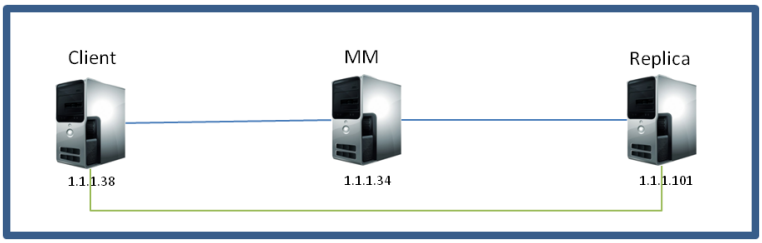
5.2.3 Description of the Experimental Setting

Several experiments have been done in order to compare the different configuration in our system, study the different results and achieve a conclusion form the results.

The *first experiment* is not going to have all the system fully integrated. We want to observe the performance offered by the system in the event that we only have a partition with a single replica at the core level. This means that the database is found in its entirety on the replica and will itself be responsible for executing all transactions, so this is our worst

case which is going to serve us as a reference for the rest of experiments. A visual example of the scenarios is shown in Figure 5.2.

Figure 5.2: First Experiment: One replica supporting all the workload



The following experiments support and the proposed structure in Figure 5.1, are going to make a difference in the type of replication protocol, see section 3.4, that will be running on each of the partitions. There have been distinguished four different experiments, although all will have in common that they will have on the second partition always the Primary Copy replication protocol. This is because having a single replica at the core level and knowing the characteristics of this protocol fits perfectly with the partition's structure, where the replica in the core level acts as the primary replica and the replicas in the first level are going to act as backups:

- *Second Experiment:* All partitions will be running the Primary Copy replication protocol.
- *Third Experiment:* Partitions 1 and 3 will be running the Active replication protocol.
- *Forth Experiment:* Partitions 1 and 3 will be running the Certification-Based Replication protocol.
- *Fifth Experiment:* Partition 1 will be running the Active replication protocol and partition 3 the Certification-Based one.

5.3 Results and Discussion

The previous experimental setting is going to be tested against our Zoo system with the two aforementioned workloads. The ZooClient Module (see Subsection 3.2.1) is configured to generate ten clients with six tasks and each of this task is going top launch, in intervals of 100 seconds, transaction with rates of 5-10-15-20 transactions per second (TPS). In Subsection 5.3.1 the result obtained from the experiments are going to be exposed and in Subsection 5.3.2 the results are going to be discussed.

5.3.1 Results

Worst Case Scenario

Let us look at the results obtained from the different experiments in the worst case workload configuration shown in Table 5.1 and 5.2:

Table 5.1: TPS Achieved vs. TPS Configured with the Worst Workload

tps	1th Experiment	2nd Experiment	3th Experiment	4th Experiment	5th Experiment
50	20	45.1	49.4	38.8	48.5
100	20	48.8	50.2	60	90
150	20	61.4	89.2	90	91
200	20	43	78	100	99.7

Table 5.2: Response Times per TPS with the Worst Workload

tps	1th Experiment	2nd Experiment	3th Experiment	4th Experiment	5th Experiment	Ideal Response Time
50	407.52	11.18	56.87	64.66	63.77	200
100	406.63	181.52	108.56	105.54	106.49	100
150	407.28	141.57	101.09	99.4	114.73	66.7
200	407.04	202.33	101.97	93.78	110.3	50

Best Case Scenario

Now we are going to present in Tables 5.3 and 5.4 the results obtained with the best case workload:

Table 5.3: TPS Achieved vs. TPS Configured with the Best Workload

tps	1th Experiment	2nd Experiment	3th Experiment	4th Experiment	5th Experiment
50	44.8	48.2	45.6	43.6	44.1
100	70	90.1	89.8	72	90
150	70	145.1	137	143.3	133
200	70	166.9	163.2	176	168

Table 5.4: Response Times per TPS with the Best Workload

tps	1th Experiment	2nd Experiment	3th Experiment	4th Experiment	5th Experiment	Ideal Response Time
50	44.5	27.66	28.99	27.99	31.52	200
100	11.73	30.81	33.8	27.72	37.26	100
150	123.97	34.86	33.5	27.59	37.26	66.7
200	126.43	34.78	39.97	26.52	34.01	50

5.3.2 Discussion

In spite of being still a prototype, the results allow us to infer the behavior of the Zoo system. We can see that in the worst case scenario the system behaves pretty decent up to 100 TPS

whereas the single node obtained 20 TPS. Another important aspect to consider here is that the replication cluster configuration matters in terms of the replication, the number of replicas and layers; as it is shown in the second experiment in Table 5.1. Keep in mind that this workload is specially tricky as it does not repeat accesses to data items. Besides, we are demanding that 80% of the transactions access to the core layer which increases the load supported by this layer and makes the rest of layers almost useless. Thus, the best case scenario represents a predominant read-only workload that can benefit from the use of the different layers. Actually, as expected, there is no difference among the replication protocol running on each partition as the update workload is mostly negligible and can be run on a single replica.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This system, that is being developing is so there are a lot of things to expand. Some conclusions obtained from this project are:

- The client developed by Mariela Louis in her Master Project [29] has been improved in order to contemplate more parameters in the transactions.

- The MM developed by Itziar Arrieta in her Master Project[30] has been improved. It has been separated from the client physically, and a module has been implemented in order to lockup through the structure that stores the partitions and the records of each one.

- It has been improved the transactional support, at the beginning only one operation per transaction was allowed, now it is possible to have as much as we want.

- Several parameters have been added to the transactions in order to get better load balance, such as the freshness level of the transaction.

With the implementation of different protocols we have achieved that system can execute several protocols concurrently in different partitions. Doing it in a transparent interface to the client an MMM. In addition with the different experiments we have studied the protocol that bets fits with each partition architecture. And thanks to Mariela Louis' Master Project [29] there was developed a module that allows to study the load in the system and select the protocol that best fits with it.

6.2 Future Work

As it has been mentioned previously, this project is being developed yet. There are several aspects that should be added to the systems, here are some of them:

- **Improving the MMM:**

  Fault tolerance: Currently, the MMM represents a single point of failure in the system. The straightforward solution is to replicate it among several replicas. As the amount of updates performed in the MMM are not so high nor so frequently executed, we will suggest the use of the PAXOS protocol [6] to replicate the MMM. Therefore, there would be a single MMM representative for managing the system with several backups in case of its failure.

  Communication interface: Each time the client needs to know the replicas where the transaction must be executed, it has to ask the MMM. This introduces higher latencies during the execution of transactions. To avoid this undesirable behavior, we propose that the client receives the PDT the first time it requests the execution of its first transaction. This information will be cached in the client throughout all of its lifetime. Once a cache fail is detected, the client will request a new PDT to the MMM.

- **Propagation Techniques:** Right now, the systems builds a propagation tree for each partition.. This is against the dynamic nature of the system and we should include more dynamic propagation techniques like communications based on epidemic propagation [27].

- **Failure and Recovery of Replicas:** The system should include a module to monitor and manage the failure of replicas as well as the addition of new ones if the workload requires to do so. We think that we should consider live migration techniques to perform this task [19].

- **Adaptation to new Partitioning Schemes:**

  Insertion of new values: The main goal of this modification is to provide the system with a new feature that lets the system to tolerate the addition of new values to the data repository. We have to modify the MMM to include in the PDT these changes (see 4.4).

  Partitioning reconfiguration: Using Mariela Louis' Master Project [29], we can infer new data access patterns using unsupervised techniques and define new data partitions and reconfigure the replica clusters accordingly.

- **Studying critical data in applications [25]:** We want to sub-partition the data associated to each partition by analyzing their nature (i.e., how critical they are to the application). Data that is considered critical will reside in the core while non-critical data will have its associated core in a replica in the outer layers; these modifications have been partially accomplished. The idea is to associate a critical level to each transaction. The updates of non-critical transaction will go to the new core layer while critical ones will go in the usual way. This feature alleviates the core from executing

all the updates. Thus, transactions with a higher critical level are executed before non-critical transactions.

Chapter 7

Curriculum Vitae

Ainhoa Azqueta Alzúaz
June 2013

Education

- Next year: Master Universitario en Software y Sistemas.
  Universidad Politcnica de Madrid (Madrid, Spain).

- September 2011- Present: Ciclo Superior en Ingeniera Informtica.
  Universidad Pblica de Navarra (Pamplona, Spain).
  Title: Landing Transactional Support on the Cloud.
  Advisors: Jos Enrique Armendriz Iigo and Joan Navarro Martn.

- September 2007 July 2011: Ingeniera Tcnica en Informtica de Gestin.
  Universidad Pblica de Navarra (Pamplona, Spain).
  Title: Cloud Computing Keeps Financial Computation Simple.
  Advisor: Jos Enrique Armendriz Iigo.
  Grade: 10 (Matrcula de Honor).

Work Experience

- Research Assistant in a National Research Project (open call).
  Research Group: Grupo de Sistemas Distribuidos.
  Institution: Universidad Pblica de Navarra.
  Date: 01/08/2012 - 31/12/2012 (5 months)
  City: Pamplona, Spain

Publications

- Mariela J. Louis-Rodríguez, Andreu Sancho-Asensio, Joan Navarro, Itziar Arrieta-Salinas, Ainhoa Azqueta-Alzúaz and J.E. Armendáriz-Iñigo. *A Prospective View on Designing Partitioning Schemes for Replicated Databases.* XXI Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2012). June 19 - 21, 2013, San Sebastián, Spain. Conference Proceedings. (Not reviewed. Accepted for publication)

- M.J. Louis-Rodríguez, J. Navarro, I. Arrieta-Salinas, A. Azqueta-Alzúaz, A. Sancho-Asensio and J.E. Armendáriz-Iñigo. *Workload Management For Dynamic Partitioning Schemes in Replicated Databases.* The 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013), May 8 - 10, 2013, Aachen, Germany. Conference Proceedings, INSTICC Press.

- Jon Legarrea, J.E. Armendáriz-Iñigo, José Ramón González de Mendívil, A. Azqueta-Alzúaz, M.J. Louis-Rodríguez, I. Arrieta-Salinas, and Francisco Daniel Muñoz-Escoí. *Boosting Performance and Scalability in Cloud-Deployed Databases.* The 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013), May 8 - 10, 2013, Aachen, Germany. Conference Proceedings, INSTICC Press.

- J. Navarro, A. Azqueta-Alzúaz, Pablo Murta, J.E. Armendáriz-Iñigo. *Cloud Computing Keeps Financial Metrics Computation Simple.* The 6th International Conference on Software and Data Technologies (ICSOFT 2011). July 18-21, 2011, Seville, Spain. Conference Proceedings. INSTICC Press.

Bibliography

[1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 159–174, New York, NY, USA, 2007. ACM.

[2] José Enrique Armendáriz-Iñigo, A. Mauch-Goya, José Ramón González de Mendívil, and Francesc D. Muñoz-Escoí. Sipre: a partial database replication protocol with si replicas. In Roger L. Wainwright and Hisham Haddad, editors, *SAC*, pages 2181–2185. ACM, 2008.

[3] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[4] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting microsoft sql server for cloud computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1255–1263, Washington, DC, USA, 2011. IEEE Computer Society.

[5] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[6] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[8] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *SoCC*, pages 143–154. ACM, 2010.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[12] Carlo Curino, Djellel Eddine Difallah, Andrew Pavlo, and Philippe Cudré-Mauroux. Opltbenchmark, 2013.

[13] Carlo Curino, Evan P. C. Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: a database service for the cloud. In *CIDR*, pages 235–240, 2011.

[14] Carlo Curino, Evan P. C. Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: a database service for the cloud. In *CIDR*, pages 235–240. www.cidrdb.org, 2011.

[15] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: an elastic transactional data store in the cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[16] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB*, pages 715–726. ACM, 2006.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[18] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[19] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD Conference*, pages 301–312. ACM, 2011.

[20] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182, 1996.

[21] Hadoop. Hbase: Bigtable-like structured storage for hadoop hdfs, 2013.

[22] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[23] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[24] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[25] Jon Legarrea, José Enrique Armendáriz-Iñigo, José Ramón González de Mendívil, Ainhoa Azqueta-Alzúaz, Mariela J. Louis-Rodríguez, Itziar Arrieta-Salinas, and Francesc D. Muñoz Escoí. Boosting performance and scalability in cloud-deployed databases. May 2013.

[26] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and José Enrique Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 34(2), 2009.

[27] Miguel Matos, Valerio Schiavoni, Pascal Felber, Rui Oliveira, and Etienne Riviere. Brisa: Combining efficiency and reliability in epidemic data dissemination. In *IPDPS*, pages 983–994. IEEE Computer Society, 2012.

[28] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In Priya Narasimhan and Peter Triantafillou, editors, *Middleware*, volume 7662 of *Lecture Notes in Computer Science*, pages 456–475. Springer, 2012.

[29] Mariela Louis Rodríguez. Estudio de modelos de predicción de consultas concurrentes en bases de datos distribuidas. Trabajo Fin de Máster, 2012.

[30] Itziar Arrieta Salinas. Study and development of a transactional database system on a cloud environment. Trabajo Fin de Máster, 2012.

[31] Nicolas Schiper, Rodrigo Schmidt, and Fernando Pedone. Optimistic algorithms for partial database replication. In Alexander A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 81–93. Springer, 2006.

[32] Damián Serrano, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *PRDC*, pages 290–297. IEEE Computer Society, 2007.

[33] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards elastic transactional cloud storage with range query support. *Proc. VLDB Endow.*, 3(1-2):506–514, September 2010.

[34] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *Knowledge and Data Engineering, IEEE Transactions on*, 17(4):551–566, 2005.