Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementing a Distributed Architecture on AngraDB

Ismael C. Medeiros

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementing a Distributed Architecture on AngraDB

Ismael C. Medeiros

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof.a Dr.a Genaina N. Rodrigues
CIC/UnB

Prof. Dr. Marcos F. Caetano
CIC/UnB

Prof. Dr. José E. G. de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 31 de janeiro de 2019

Dedicatória

Este trabalho é dedicado a todos as pessoas que me ajudaram nestes últimos cinco anos a passar por esta fase turbulenta que foi a minha graduação. Também dedico as pessoas que nesse tempo me ajudaram a estar sempre me melhorando, seja pela crítica ou pelos grandes conselhos.

Agradecimentos

Primeiramente, eu agradeço a minha família pelo suporte em todos esses anos. Aos meus pais, Manoel e Vera pelo ensinamentos passados no percorrer da minha vida. Também agradeço aos meus grandes amigos, Iure, Iago, Yuri, Gustavo, e Gabriel. Vocês estiveram comigo durante todos esses anos e continuarão para resto da vida. Em especial, meus agradecimentos vão para os meus amigos Iure e Iago, que mesmo não dando tanto tempo para revisar este projeto, se prontificaram a me ajudar.

Agradeço aos meus amigos que tive o prazer de conhecer na faculdade, Bruno, Lucas, Amanda, Ingrid, Claudio, Edgar e aqueles que não citei para a lista não ficar muito grande. Vocês acrescentaram muito a minha vida. Também, este agradecimento vai às grandes pessoas que conheci trabalhando na Lacuna Software, Daniel, Abilio, Leo, André, Bruno, Pedro, Lydia, Alexandre, entre outros. Vocês são pessoas muito boas no que fazem e sempre me inspiraram a melhorar. Também vocês me mostraram as partes boas da área de computação, sou muito grato a vocês por isso.

Agradeço aos meus parceiros neste projeto do AngraDB, Bruno, Fabio, e Fernado, vocês fizeram com que esse projeto se tornasse mais um prazer do que uma simples obrigação. Também agradeço nosso orientador e líder do projeto, o professor Rodrigo Bonifácio, pelas dicas e pelo suporte dado em todos esses anos de projeto.

Por fim, agradeço a comunidade do StackOverFlow, que incansalvemente, tem acrescentado muito conhecimento relacionado a problemas de programação, inclusive em Erlang. Também, meu agradecimento vai aos moderadores do Erlang Solutions, que possui artigos e serviços bastante úteis para este projeto.

Resumo

Big Data refere-se à quantidade crescente de dados consultados e processos nos mais diversos formatos (v.g., textual, imagem, vídeo) e domínios (v.g., streaming the conteúdo, serviços de blockchains). Para poder se alinhar com os desafios do Big Data, provedores de serviços conhecidos começaram a desenvolver e usar diferentes modelos de banco de dados (incluindo banco de dados, como Bigtable, orientados a documentos e a chave-valor). A implementação de um banco de dados é uma tarefa desafiadora, que permite aos alunos explorarem diferentes áreas da Ciência da Computação. Por esta razão, e com o objetivo de melhor entender e disseminar o uso da linguagem de programação Erlang, iniciamos o desenvolvimento do AngraDB, um banco de dados orientado a documentos de código aberto. Neste trabalho, contribuímos com um recurso específico e relevante que um banco de dados deve suportar: distribuição para alto desempenho e disponibilidade. Para tanto, foram investigadas (a) as técnicas que outros bancos de dados usam frequentemente para a replicação e consistência dos dados, (b) protocolos existentes para sistemas distribuídos, e (c) a arquitetura Erlang para implementação de sistemas distribuídos. Em seguida, projetamos e implementamos a arquitetura distribuída do AngraDB, validando a arquitetura através de um estudo de caso.

Palavras-chave: Distribuição, banco de dados, arquitetura, tratamento de erros, configuração, disponibilidade

Abstract

Big Data relates to the increasing amount of data queried and processed in the most different formats (e.g., textual, image, and video) and domains (e.g., content streaming, blockchain services). To cope with Big Data challenges, well-known service providers started to develop and use different database models (including Bigtable, document-oriented, and key-value databases). Implementing a database is a challenging task, which enables students to explore different Computer Science areas. For this reason, and aiming to better understand and disseminate the use of Erlang programming language, we started the development of AngraDB, an open-source, document-oriented database. In this work, we contribute with a specific and relevant feature that a database must support: distribution for high performance and availability. To this end, we investigated (a) the techniques databases often use for data replication and consistency, (b) existing protocols for distributed systems, and (c) the Erlang architecture for implementing distributed systems. After that, we designed and implemented a distributed architecture for AngraDB, validating the architecture through a case study.

Keywords: Distribution, database, architecture, error-handling, configuration, availability

Contents

1 Introduction **1**
1.1 Problem . . . 2
1.2 Goals . . . 2
1.3 Organization . . . 3

2 Theoretical Basis **4**
2.1 CAP Theorem . . . 5
2.2 Dynamo Paper . . . 7
2.3 Case Studies . . . 8
2.3.1 CouchDB . . . 8
2.3.2 Riak . . . 9
2.3.3 MongoDB . . . 10

3 Distributed Algorithms **12**
3.1 Consistent Hash . . . 12
3.1.1 Cryptographic Hash Function . . . 12
3.1.2 Ring Architecture . . . 14
3.1.3 Sharding Strategy . . . 14
3.1.4 Replication Strategy . . . 15
3.1.5 Handoff Operations . . . 15
3.1.6 Virtual Nodes . . . 16
3.2 Gossip Protocol . . . 17
3.2.1 Dissemination . . . 17
3.2.2 Anti-entropy Algorithm . . . 19
3.2.3 Triggering *Sync* . . . 19
3.2.4 Usage . . . 20
3.3 Sloppy Quorum and Hinted Handoff . . . 20
3.4 Vector Clocks . . . 21

4 Implementation **22**
4.1 Erlang . . . 22
4.1.1 Actor Model . . . 22
4.1.2 Distribution . . . 23
4.1.3 Behaviors . . . 24
4.2 Application Architecture . . . 24
4.2.1 The *adb_gossip_** Modules . . . 25
4.2.2 The **_partition* Modules . . . 25
4.2.3 The *adb_vnode_** Modules . . . 27
4.2.4 The *adb_dist_** Modules . . . 28
4.2.5 The *adb_server_** Modules . . . 28
4.2.6 The **_persistence* Modules . . . 28
4.2.7 Design for Configuration . . . 29
4.3 Cluster Architecture . . . 29
4.3.1 Membership . . . 30
4.3.2 The *ring_id*'s Assignment Strategy . . . 30
4.3.3 Highly Available Server . . . 31

5 Experiments **33**
5.1 Environment . . . 33
5.2 Local Distribution with Docker Compose . . . 34
5.3 Testing Consistent Hash . . . 35
5.4 Testing Gossip Protocol . . . 38

6 Conclusion **39**
6.1 Learned Lessons . . . 39
6.2 Future Works . . . 39

References **41**

List of Figures

2.1 The CAP theorem's choices made by some known databases. 6
2.2 Riak Throughput vs. Number of Nodes. 9
2.3 Representation of the MongoDB's replication strategy. 10

3.1 Sharding and replication of keys in Dynamo ring. 14
3.2 Riak's Ring Architecture. 16

4.1 AngraDB Architecture Diagram. 24

5.1 Hypervisor-based and Container-based Virtualizations. 34
5.2 Load distribution when using MD5 in a three-nodecluster. 35
5.3 Load distribution when using MD5 in a four-node cluster. 36
5.4 Load distribution when using SHA-1 in a four-node cluster. 36
5.5 Load distribution when using SHA-256 in a four-node cluster. 37
5.6 Load distribution when using SHA-384 in a four-node cluster. 37
5.7 Load distribution when using SHA-512 in a four-node cluster. 37

List of Tables

2.1 Summary of techniques used in DynamoDB. 8

3.1 Summary of cryptographic hash functions. 13

4.1 Ring configuration according to the number of nodes on the cluster. 31

Acronyms

ACID Atomicity, Consistency, Isolation, Durability.

AP Availability and Partition Tolerance guarantees.

API Application Program Interface.

BEAM Bogdan's Erlang Abstract Machine.

CA Consistency and Availability guarantees.

CAP Consistency, Availability, Partition Tolerance.

CP Consistency and Partition Tolerance guarantees.

EPMD Erlang Port Mapper Daemon.

JSON JavaScript Object Notation.

MD5 Message-digest Algorithm 5.

OTP Open Telecom Platform.

P2P Peer-to-Peer.

REST Representational state transfer.

SHA Secure Hash Algorithm.

SQL Structured Query Language.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

VM Virtual Machine.

Chapter 1

Introduction

Over the past few years, the amount of information produced has increased so much that the available storage technologies became obsoletes. Further, the volume of data is still expanding much faster than before[1], leading to the *Big Data* concern[2]. The lack of well-defined patterns presented in this volume of the data caused new technologies to be developed to fulfill the gap that traditional systems could not.

In this context, different alternatives to organize data have been proposed, those that do not get stuck on rigid patterns and support all kinds of information. A new kind of database had arrived, it was called NoSQL in opposition to the existing databases [1]. However, we have to emphasize that NoSQL embraces many more kinds of stores, including relational databases (for that fact, it began to be interpreted as "Not Only SQL" instead of "Not SQL"). NoSQL stores can be categorized [2] based on the chosen data model in:

- Wide-column stores (e.g. Hadoop and Cassandra);
- Document databases (e.g. CouchDB and MongoDB);
- Key-value stores (e.g. Riak and DynamoDB);
- Graph-oriented stores (e.g. Neo4j); And
- Others kinds of store (e.g. multi-model, objects and multi-dimensional arrays).

In 2016, the AngraDB [3] project begun. Initially, the main goal was to learn the programming language Erlang and immerse in a challenging project like the development of a database. This language belongs to the functional paradigm and has great support for distribution and error handling. We defined a set of goals, which approached different

[1] By 2020, it is estimated that, in every second, 1.7MB of data will be produced on average by each person on earth (Source: `https://www.domo.com/learn/data-never-sleeps-6`)

[2] Described as the search for strategies to efficiently process great volumes of data with multiple formats

fronts, such as the database's file system, the search and indexing mechanism, client libraries, and a REST interface. Although the available Erlang support for building distributed systems, the first versions of AngraDB did not implement advanced distributed features, which are typically found on NoSQL databases.

1.1 Problem

The capability of handling infrastructure problems (e.g. the crash of the host that runs the application or it becomes inaccessible because of network problems) is a highly required property in databases. Also, at the same time, the database cannot lose considerable availability rate of its service, even if the database is suffering a heavy load of requests. This need is caused by the growing demand by applications, where data access is a critical asset. Further, in a time where information is a becoming currency, databases are being more and more demanded. A reliable application has to handle these tasks. However, as mentioned before, the previous versions of AngraDB do not support those relevant features, such as high availability and fault-tolerance. In order to provide a reliable service, the database has to use several techniques to create an architecture capable of handling real-world tasks.

1.2 Goals

When we are developing an application, that must be capable to process so many requests that it does not have the capability to do so, the first thing to do to resolve is scaling the application. There are two kinds of scalability [4]: scale-up (vertical) and scale-out (horizontal). Scaling-up is related to adding more resources to a single host in order to improve the performance of the system. However, this improvement is limited by the current technology available to add to the system. On the other side, scaling-out is referred to adding more hosts in order to form a cluster[3] and to divide the load between nodes[4]. However, as mentioned before, the previous versions of AngraDB do not support those relevant features, such is high availability and fault-tolerance. In order to provide a reliable service, the database has to use several techniques to create an architecture capable of handling real-world tasks.

In order to achieve high availability, the data storage has to be decentralized, in order to not letting the whole system to become vulnerable to eventual single points of failure[5].

[3]In distributed computing, it refers to the set of hosts, whose form a single system by working together

[4]Refers to one of the hosts that compose the cluster

[5]A portion of the application that can compromise the whole system in the case of errors

This will be accomplished with the distribution of the database's storage structure, and for that, it will be necessary to design a resilient architecture for the system.

Thus, this work focuses on the implementation of the algorithms, which have already been consolidated into other systems and provide support for decentralized storage:

- Consistent Hash [5];
- Gossip Protocol [6];
- Handoff Operations[6];
- Strategies for Failure Detection and Correction [7]: Sloopy Quorum, Hinted Handoff, and Vector Clocks.

After developing those items above, they should be tested, in a way to guarantee the correctness of the implementation. This work will focus on the behavior presented by the database in an optimistic scenario, where it is trusted that no errors will happen. On the other side, the auto-healing capabilities provided by the distributed architecture will not be tested, but it is planned as future work.

Thus, the contribution of this work is to investigate existing approaches to design and implement a distributed architecture on top of AngraDB.

1.3 Organization

This work is organize in a way to present theoretical aspects first in order to present the reader a background before explaining the implementation aspects of this project.

In chapters 2 and 3, some of the main concepts involving distributed systems will be presented in details and several techniques are exposed, in order to understand their importance in distributed applications. In chapter 4, the design choices and implementation aspects will be shown. In there, the planned architecture and how the Erlang characteristics have influenced on this architecture are presented. The main tools provided by this language is shown in this chapter. At last, chapters 5 and 6 present the performed experiments and the conclusions about the results of this work.

[6]A Handoff operation treats the document exchanges when some node is entering or leaving the system

Chapter 2

Theoretical Basis

In this chapter, some concepts about distributed computing are discussed. First, we will present the CAP theorem [8], which is one of the most important concepts of distribution. It has served as the foundation for the decision making on many distributed file systems. Also, a study about the Dynamo paper [7] had to be done to understand how it became such an influence in the choices made by other databases. Lastly, some decisions made by important distributed databases are presented. These choices had to be comprehended in a way to understand what problems they are supposed to solve. The analyses shown in this chapter was valuable on the choices made while planning a design for AngraDB.

A distributed system presents features that help to provide reliable service while handling errors. We can quote two of the most important between them: the decentralization and the elasticity. A decentralized system has the leverage to present fewer points of failure than usual applications. Therefore, each node must be capable to exercise different roles in the cluster, in order to reduce the dependencies between nodes and tasks. Thus, the system will not rely on a single node alone to handle a critical task.

Additionally, the elasticity has the same importance to the failure tolerance in an application than the decentralization. This property is directly related to the system's capability of handling the gaps left by a crashed node. Also, the flexibility is related to the capability of handling new nodes entering on the cluster. In situations they had to reorganize in a way to handle failures, systems with this property, have to handle newcomers with the same purpose, to adapt to the new environment.

When we talk about decentralized systems, the state stocked by a host might be an issue. If other hosts need to access this information, they become depended to that node. This creates a new point of failure. Usually, when we distribute an application, we divide the application between business logic and user interface. The business logic is always

unique in the whole system, in order to guarantee consistency[1] in multiple instances of the system. Only the user interface is replicated to provide high availability.

However, databases must always carry a valuable volume of data, so the same architecture could not be applied. It is necessary to spread its state into different hosts in order to guarantee failure tolerance in the case of errors. This can provide a backup when some node is unavailable. But if the data is replicated into multiple nodes, it causes another problem, it becomes more difficult to guarantee that all hosts stay updated[2]. On the other side, if the system's designer chooses to provide more consistency by reducing the copies of each information, the capability of handling errors can be impaired.

This is an important example of trade-off when drawing a distributed architecture for a database. And had served as the inspiration to the concept of the CAP theorem.

2.1 CAP Theorem

In 2000, Eric Brewer has brought up a theorem about design choices when creating a robust distributed file system [8]. It is known as "CAP Theorem" and states that a distributed storage system can only guarantee at most two of the following properties:

- Partition Tolerance
- Consistency
- Availability

The first property "partition tolerance" refers to the capability of the system to handle network partition. When building a distributed application, it's necessary to expect that eventually, some node would try to contact another and fail. This can be caused by network problems, such as high latency, low bandwidth, and other common issues, or caused by the target node's crash. To handle these problems, the system must provide mechanisms that will reorganize the database in order to fill the gaps left by the inaccessible node. Thus, as mentioned before, it is well-known that it is essential to have this property to provide elasticity in the face of errors. The guarantee to always be partition-tolerant is the most important guarantee in the CAP theorem to a distributed system.

The second property "consistency" refers to the responsibility that the system has of providing the newest data to the user, even in case of errors. This has to happen regarding the node, where it recovers the information. In this context, this data is referred to the last writing on the system. If the system cannot return this response, it must send an

[1]In this context, consistency is referred to the state of a cluster where all hosts have the same version of some data.

[2]This state is known as the consistency of the cluster

error to the user instead of older information. This guarantee can be confused with the "consistency" from the ACID set of properties of databases transactions. In that case, it was talking about the state of a single database and how a transaction will not bring the database's state to an invalid one. It is closely related to the database relational structure. However, distributed databases, which choose to provide this guarantee, also aim to provide the properties from ACID on its transactions. So, it is valid to say that the two guarantees are related.

The third property "availability" states that, as long as there are healthy nodes on the cluster, the system must always provide an answer to the user, even if it is not the newest one, and never return an error to the user. Commonly, this is achieved by replicating the database in order to always a have an available node to store or provide the data. This property commonly conflicts with the second (consistency), causing the choice on the CAP theorem to be reduced to a single decision between these two guarantees.

This theorem is commonly misinterpreted [9] as the not-guaranteed property is never provided by the system, but this theorem is only related to the time when errors occur. A system, that does not present errors so commonly, will not need to worry so much with the property that it does not guarantees. Also, there are techniques designed specially to work around an eventual lack of one of these properties. It is the job of the designer to predict some of these problems that might happen.

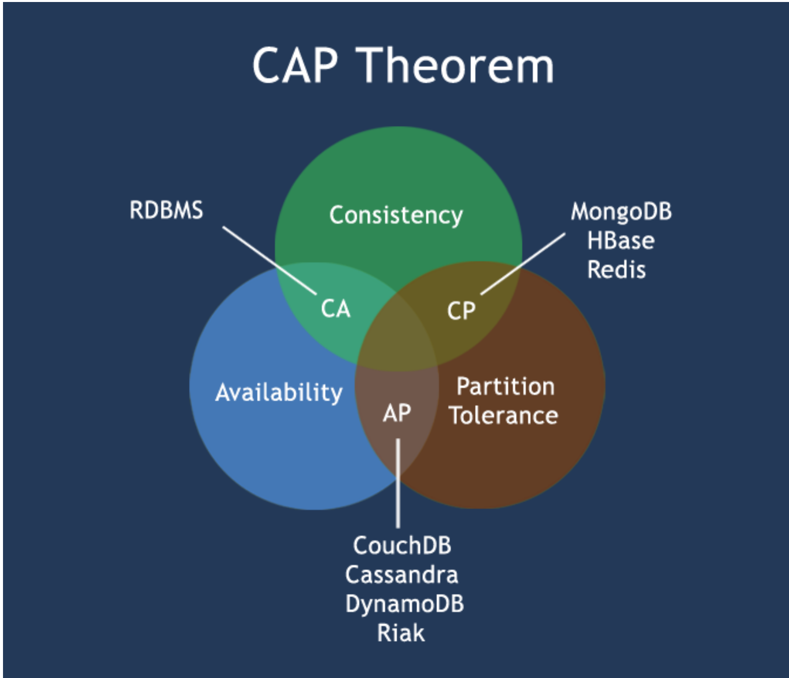

Figure 2.1: The CAP theorem's choices made by some known databases (Source: [10]).

After observing the diagram the Figure 2.1, it is remarkable that the distributed databases had chosen to provide the "partition tolerance" guarantee, as it was expected. In the other side, they are divided into two groups according to their decision between "consistency" and "availability", these are named as AP and CP databases respectively. Relational databases are isolated on the diagram, they are located on the CA databases section, because it cannot provide the capability to handle network partition without losing its characteristics of a traditional database, such as ACID transactions and consistency.

2.2 Dynamo Paper

In 2007, Amazon made available an article [7] describing one of their web services, it was about a database called Dynamo. This database was considered essential to their e-commerce service, where an instance of unavailability could result in millions lost. The eventual consistency could be handled, but a not excellent available rate could be catastrophic.

DynamoDB was designed as a key-value store, that provides the guarantees of "availability" and "partition tolerance". One of the main features of this database is that it focuses in provide high availability on writing operations. In an e-commerce system, a single error or delay on the website can make a potential client give up a purchase, but, in a system with the massive proportion like the Amazon's, it is really difficult to provide a highly available service. The Dynamo's main goal, as described in the article, is to provide a 99.9% availability rate[3] on writing operations.

As an AP distributed database, DynamoDB has to provide the mechanisms to achieve its desired goals. In the articles, there was listed a set of problems about distribution and what technique was used to solve them, the same summary is exposed in Table 2.1.

These techniques are explained with more details on Chapter 3, where their aspects are discussed and comprehended, in order to observe their importance in a distributed database.

After its publication, many designers based on this article and the DynamoDB specifications served as the basis to the creation of other distributed databases. We can quote two of them, which were used and are still being used as references in the implementation of AngraDB, these two are CouchDB and Riak. Both databases are distributed but have taken different aspects in resolving its problems. In the following section, we report our understanding of the main design decisions related to distribution taken during the

[3]Equivalent to 8.77 hours per year, considering that a year has 365.25 days.

Table 2.1: Summary of techniques used in DynamoDB (Source: [7]).

Problem	Technique
Partitioning	Consistent Hashing
High Availability for writes	Vector clocks with reconciliation during reads
Handling temporary failures	Sloppy Quorum and hinted handoff
Recovering from permanent failures	Anti-entropy using Merkle trees
Membership and failure detection	Gossip-based membership protocol and failure detection

implementation of CouchDB, Riak, and MongoDB. These decisions influenced the design and implementation choices of AngraDB.

2.3 Case Studies

This section shows three case studies about the following NoSQL databases: CouchDB, Riak, and MongoDB. As mentioned before those are distributed and had served as models on the AngraDB's design decisions.

2.3.1 CouchDB

CouchDB is an open-source AP document database written in Erlang and served as great influence on AngraDB project, not only on distribution aspects. This study talks about aspects found out on its official documentation [11] and open-source code provided by Apache. On a CouchDB cluster, nodes communicate with each other using a REST interface and carry a database called _replicator, which is responsible for processing the CouchDB Replication Protocol.

In this protocol, every writing operation will cause that a new replication document enters on the _replicator database, causing that the replication procedure starts. A node with a started procedure will contact another node with changes made, causing the contacted node will be updated with the new data, generating a new entry on its _replicator database and eventually contacting another node and so on. Also, this database has the responsibility of storing partial changes to prevent that whole operation fail caused by an error on the target node, it will store a partial state to be used later to restart the replication process. This kind of replication is performed in one direction and it is called "Incremental Replication". This replication capability is treated with priority and it is one of the most important features of CouchDB.

Influenced by DynamoDB, this database also uses consistency hash for sharding[4] and uses a configurable quorum for validating an operation on the cluster. Besides the replication aspects, another CouchDB's feature is the fact of the communication between nodes is made by REST APIs. This facilitates that nodes, which was not been implemented in Erlang, could interact to the system as nodes. There is one implementation in JavaScript called PouchDB, that behaviors as a typical node on the system, and serves as an offline cache to the client application, executing within the browser.

2.3.2 Riak

Riak is an AP key-value database, that inherits from DynamoDB most of its features and its main techniques, such as Gossip Protocol, Consistent Hash, Sloppy Quorum and Vectors Clocks [12]. It is written in Erlang and takes advantage of the RELEASE project, which aims to improve the scalability of Erlang on architecture with 10^5 cores [13].

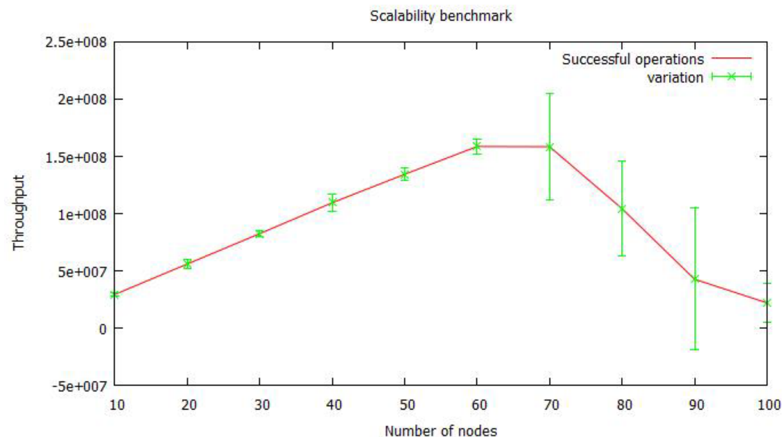

Figure 2.2: Riak Throughput vs. Number of Nodes (Source: [13]).

The database's scalability was investigated by researchers from the University of Glasgow [13]. After comparing Riak with other databases written in Erlang, they performed a benchmark with the goal of analyzing the throughput with a different number of nodes on the cluster. As can be seen in Figure 2.2, the database has behaved well in presence of so many nodes working together, increasing its throughput until its peak at $\approx$ 65 nodes. Besides, it was informed that Riak has shown great elasticity in face of errors during the benchmark.

As AngraDB's distribution architecture has been mostly influenced by the principles on the Dynamo paper, this analysis was very important to verify the behavior of the

[4] A sharding operation refers to the fragmentation of a database, storing documents on different nodes, according to the key mapping strategy, instead of replicating the document on every node.

DynamoDB's features written in Erlang and to analyze how much nodes the system support without losing performance.

2.3.3 MongoDB

In spite of the previous databases, MongoDB chooses to guarantee "consistency" instead of "availability", it was created inheriting from relational databases some consistency features, in especial the ACID transactions' philosophy, but without losing the flexibility of a NoSQL and the capability of providing support in modern applications. One of its many critics about the eventual consistency model is that this kind of consistency can turn the developer's life very difficult because it has to handle with multiple version of the same document.

MongoDB is ACID compliant at the document level, ensuring complete isolation on document operations, in special the write operations. Any error causes that the operation to rollback and clients to receive a consistent view of the database. The database's "consistency or availability" trade-off is defined mainly by its replication strategy.

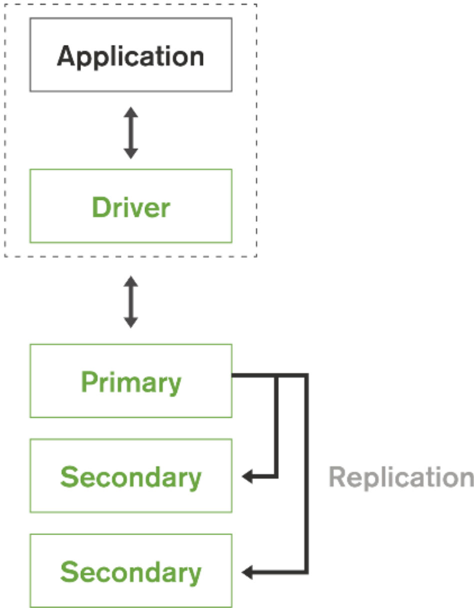

Figure 2.3: Representation of the MongoDB's replication strategy (Source: [14]).

In Figure 2.3, there is a representation of how replications are done at MongoDB. Each data is mapped to one primary node and to guarantee high availability in presence of errors, the information is copied to a number of secondary nodes. By default, the database chooses strong consistency, where every writing and especially reading operations are made exclusively on the primary node. When this node become temporary inaccessible

or crashes for some problems (e.g., hardware failures and network partition), the secondary nodes will elect a node to be the primary until the first get operational again, and at this moment, it will acquire all updates made at the secondary node in the time it was out of the cluster.

MongoDB is flexible to the point of prioritizing availability against consistency, becoming an AP database, by allowing readings in the secondary node even in the case of the primary is healthy. This can be adjusted on the database's configuration, this is important in cases where it is necessary for the application to access the closest node to get some data, reducing the geographic latency, and the application does not need the most up-to-date data.

While analyzing its architecture on MongoDB's white paper [14], it was possible to see some similarities with AngraDB, especially at the flexibility of configuration. When creating an instance of this database, the partitioning policies can be chosen between the following three types:

- Range Sharding: The keys are mapped based on their values. In this case, the ranges are predefined.
- Hash Sharding: The keys are mapped based on the hash value of each key. The ranges are defined on the consistent hash algorithm.
- Zone Sharding: The keys are mapped based on their document's type. Each key are mapped to its respective zone.

Another possible configuration is one of its most bold features, it is possible to set which database model to use. Besides the fact that the document is mainly document-oriented, its architecture provides the possibility of working with other data types, such as key-value pairs, flat or table-like structures. This is all possible because the document model presents a superset of other data models and a robust query language [5].

[5] A robust query language has to be expressive enough and to be capable of processing different kinds of data, it is essential to a NoSQL database

Chapter 3

Distributed Algorithms

This chapter describes some distributed algorithms used on the database partitioning into multiple hosts. These are responsible for providing a highly available and fault-tolerant service. Thus, the main goal here is to understand those algorithms in order to analyze their importance on the design decisions of this project.

As mentioned before in Chapter 1, we focused on the Consistent Hash algorithm and on the Gossip Protocol. They are key features to design a distributed system. So, they were chosen to be the first to be implemented on the top of AngraDB.

Then, we have chosen an algorithm responsible for assign nodes to store a range of documents. This will help to divide the load into the cluster. Also, it has a strategy to handle possible errors by using redundancy. This algorithm is called "Consistency Hash".

3.1 Consistent Hash

A consistency hash is a kind of sharding algorithm, that uses a hash function to the maps keys into flexible ranges of hash values. It was originally created [5] to be used on a distributed cache in order to continue working efficiently while the topology of the network is continually changing. This algorithm is different from a common hash table, it does not use modular operation to divide the hash, in order to handle the entering and leaving of nodes from the cluster, without the need of recalculating all ranges every time this happens. This is the main cause of the elasticity of a distributed database.

3.1.1 Cryptographic Hash Function

Before understanding this algorithm more deeply, its necessary to know what a cryptographic hash function is and which features are important to the algorithm. A typical hash function is a map, which receives a variable-length input string, which is called a

Table 3.1: Summary of cryptographic hash functions (Source: [15, 17]).

Hash Function	**Output's Length**
MD5	16 bytes
SHA-1	20 bytes
SHA-256	32 bytes
SHA-384	48 bytes
SHA-512	64 bytes

"pre-image", and converts it to a fixed-length (generally smaller) output string, which is called a "hash value" [15].

An ideal hash is a deterministic injective function—that is, an input A will always be mapped to the same hash value $h(A)$. Considering the significant compression of the input, the result of a hash function can be used to identify the input's data. We can quote two very important uses of this characteristic. First, it is important on storage files, because the hash, a lighter value to be stored, can identify some stored data. It is commonly assumed [5] that a hash function maps objects uniformly and independently. This is necessary for the hash function to be used to distribute the load in a cluster. Second, hash functions are highly used on secure communication, the fact of the hash represents the content of arbitrary information can be used to validate this data's integrity during transmission. This is very important to handle network problems in a distributed system.

In the practice, there is no such ideal function, so, to be considered secure and useful, a hash function must present the following collection of properties [16]:

- Pre-image Resistance: Having a output hash value, it is computationally infeasible[1] to find any input which hashes to that output;
- 2nd Pre-image Resistance: Having a message m_1 and its hash value $hash(m_1)$. It is computationally infeasible to find any second message m_2, which $hash(m_2) = hash(m_1)$. This property is also called as "Soft Collision Resistance";
- Collision-resistant: It is computationally infeasible to find two different messages m_1 and m_2, where $hash(m_1) = hash(m_2)$.

The available cryptographic hash functions to be used on this project and their corresponding output length can be found in Table 3.1.

[1] The expression "computationally infeasible" may involve a super-polynomial effort

3.1.2 Ring Architecture

In this algorithm, nodes are organized as a ring. Each node had an essential role on the key mapping, for this, they are assigned with an identification value, that represents a point of the Hash space[2]. From now on, this identification will be named as *ring_id*.

This identity is essential to the sharding strategy, where keys are mapped to its respective node, according to its *ring_id*. The ring format is caused mainly by the chosen sharding and replication strategies, which treaties the node list with each *ring_id* as a cyclic list. These strategies are going to be explained deeply in the next topics. A famous representation of a ring architecture from the DynamoDB paper [7] is shown in Figure 3.1 to help to visualize this kind of architecture.

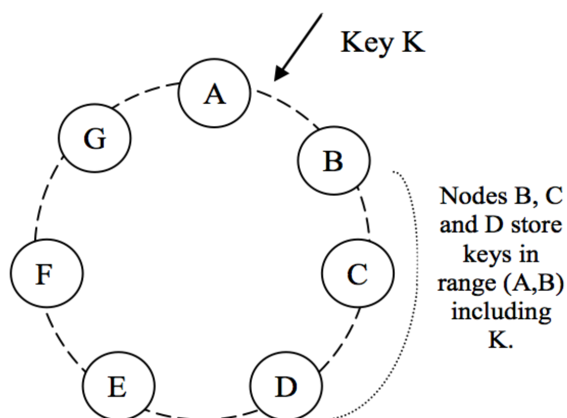

Figure 3.1: Sharding and replication of keys in Dynamo ring (Source: [7]).

3.1.3 Sharding Strategy

In a consistent hash algorithm, the sharding strategy depends mostly on the preset *ring_id*s. Every time an operation that handles a single document is requested, it's necessary to pass the key that identifies that data on the collection. So, every request of this kind, it is produced a hash of the value of this key, to be identified on a cluster level. After this, it is built a list of all nodes ordered by its *ring_id*s, to be used on the sharding. In ascending order, the *ring_id* of each element of this list is compared to the calculated hash of the key, if the hash value is less than the *ring_id* value, the key should be stored on that node, otherwise. However, if it is greater, the algorithm passes to the next element of the list to be processed until the list finishes.

[2] A hash space is the set of possible values that a hash function can produce. (e.g. MD5 can output 2^{128} values, that is about 10^{38} different values)

This algorithm can be better understood considering the following function, which is based on the original description [5] of the Consistent Hash:

function KEYSHARDING(key)
  $hashKey \leftarrow hash(key)$
  $orderedList \leftarrow getListOfNodesOrderedByRingId()$
  repeat
    $ringId \leftarrow getNextRingId(orderedList)$
  until $hashKey < ringId$
  $mapKey(hashKey, ringId)$
end function

There are some considerations about *ring_id* definitions, that will provide a good use of this algorithm. First, there must be a node that has a *ring_id* equal to the highest value that the chosen hash function can produce, to guarantee that every key will be store regardless of its value. For example, when using the SHA-1 hash function, the highest possible value is $2^{160} - 1$ (according to Table 2.1), so the cluster must have a node with this value as a *ring_id*. Secondly, it is highly recommended to not generate a *ring_id* with a value close to zero for efficiency reasons, if a node has a *ring_id* with such value, almost no document will be stored at the node on this algorithm.

3.1.4 Replication Strategy

In order to provide safety in case of errors, each document is replicated in a predefined number of adjacent nodes to serve as a backup in case of the primary node, defined by consistent hash, crashes. These copies have great value while providing high availability guarantee, but it hinders the consistency because on every request received, there are more nodes to be updated in case of a write operation. For that reason, this parameter along with the sloppy quorum, that will be explained later, are some of the main configurations on AngraDB, to give the user the possibility of tuning it. This feature was inherited from DynamoDB [7] and Riak [12], and they are commonly setting this replication value to three, i.e., every write operation will be replicated on the next two nodes on the cluster, after it executes on the target node, according to the ring architecture of the system.

3.1.5 Handoff Operations

The main reason for using consistent hash is about its benefits when dealing with handoff operations [5], which is the node entering and leaving events during execution of the system. When the system detects that a node is unavailable, it removes that node's *ring_id* and continues to work normally, every request that was supposed to be mapped

to the removed node will be going to the node with *ring_id* directly greater than the first one, no additional changes on the cluster organization are required. On the cases where a new node enters on the ring, it's generated a new *ring_id* to it and will enter the ring to be synchronized. Also, in this case, the node with *ring_id* immediately superior to the newest node will be working to synchronize the node, passing the respective documents that have to be stored on the node. In this algorithm, the handoff operation only requires that at most N/n documents to be reallocated when a node enters or leaves the cluster, where N is the maximum number of keys, and n is the number of *ring_ids* on the cluster.

These handoff operations cause only local changes on the cluster. This is great for the elasticity of the system, but it can cause load distribution issues. In scenarios where a significant amount of nodes enter or leave the cluster frequently, the key ranges, that each node is responsible, are doomed to become asymmetric, i.e., there will be nodes responsible for more keys than the others. This problem is treated by using virtual nodes, an abstraction of the physical node that will help to distributed load on the cluster in a simple way.

3.1.6 Virtual Nodes

A virtual node is a process that behaviors as a normal node (we will refer it as "physical node" for now on), it receives a *ring_id* and participates on the consistent hash normally. One physical node will contain many virtual nodes. This abstraction will cause that a physical node to be responsible for different key ranges on the cluster, improving the load distribution on the system. As a common configuration, every virtual node is positioned aside from another virtual node from a different physical node. An example of this configuration can be seen in Figure 3.2, where there is a representation of the ring architecture and virtual node distribution on Riak.

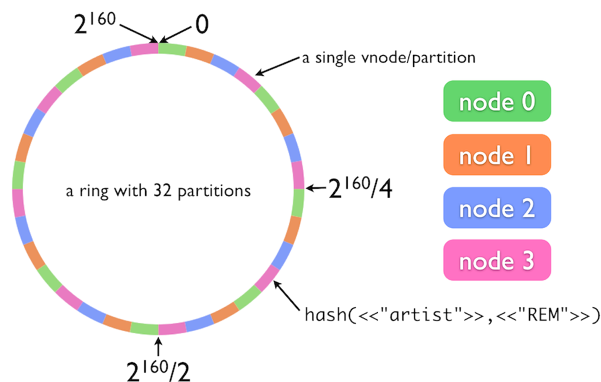

Figure 3.2: Riak's Ring Architecture (Source: [12]).

As in DynamoDB and Riak [12], the number of vnodes will be configured on AngraDB. The minimum recommended value is three because, in case of errors on one node, there will be two others left, one for writing operations and another for reading operations. These recommendations are based on analysis made by Amazon and are shown on Dynamo paper [7].

3.2 Gossip Protocol

This section presents the Gossip Protocol and what is the importance of this algorithm on a distributed system. This is an algorithm that has the purpose of disseminating information through a system, only using peer-to-peer communication.

There is a naive, but a very common approach to disseminating information in distributed systems, a node sends data to all other nodes on the system every time, through broadcasting[3]. In this approach, it does not consider the possibility of eventual errors on the network and it depends that node knows every node presented on the system, which, in a very large system, it can happen. A database, that relies on a "partition tolerance" guarantee, should not trust in the perfect network functioning.

In the presence of errors, the peer-to-peer communication is essential, mostly because it does not rely on the node knowledge of the system, for the case, it only needs to know another node. Dissemination is a special kind of algorithm based on P2P communication and is used to spread information to a distributed system with large proportions.

3.2.1 Dissemination

Based on the dissemination process presented by contagious diseases [18], those algorithms that inherit many aspects from the real process has been called as "dissemination algorithm". In the case of real propagation process, an infected agent will only transmit the condition to other after a contact happens. In a real situation, there are millions of agents that are in constant contact, making the diseases to spread at a critical speed, if not treated. On computing aspects, the disease is relative to the information and the agents are related to the nodes on the system. The information exchange between nodes will be made exclusively by P2P operations. The constant contact in a real world will be replaced by a periodic trigger on the algorithm. In this way, the data will be spread across all nodes just like in a disease situation.

The "Gossip" Protocol is an example of dissemination protocol. From its name, it is easy to discover where its inspiration is from. In a social network, information is spread at

[3]Broadcasting is a kind of network communication where an agent transmits some information to every node it can contact

a higher speed compared to the diseases dissemination process. In the era of the internet, this speed had grown up on an unprecedented scale, causing problems such as “Fake News”. The information exchange process in this kind of network can be explained by the following example:

In a social network, Alice has a list of friends that she frequently talks to. When she has some news to talk about, she likes to talk to his friends about that. In this case, Alice gets her friends list and picks one to talk about. If that person doesn’t know about the new information, Alice will be motivated to tell the novelty to another friend. If not, Alice will stop bothering his friends with out-to-dated information.

Besides the fact the information is spread in a much more messy way in a real situation, its process, described by the sample above, has inspired the Gossip Protocol because of its well-defined steps. This set of actions can be summarized in the following list:

1. Receive new information;
2. Chooses randomly a target to share;
3. Spread new information:
    - If the target does not know it, repeat these steps, choosing another target, if there is any;
    - If not, stops.

When a node is updated by another, it triggers a new *sync* operation. In the best situation, after the system stops triggering a Gossip’s sync pulse[4], and in the end, the cluster should be updated. But, there is the possibility of the update stops before the remaining nodes get updated (e.g., nodes can contact a node that was already been updated, causing them to stop. Depending on the size of the cluster, this can stop the whole pulse event). And this probability can be manipulated by using a specific anti-entropy algorithm, which will be explained later.

These steps can be translated into an algorithm, where the most important step is the comparison between the agents’ state. In this process, everyone will be updated according to the information exchange. More than one subject will be compared in this process, so the choice to continue exchanging data will involve the whole state, not just about a single subject.

[4]In this project, the term *pulse* refers to the chaining gossip interactions being activated when firstly triggered by just on node

3.2.2 Anti-entropy Algorithm

In the dissemination, it is important to set an anti-entropy algorithm, which is a strategy to compare data when exchanging information with another agent [18]. And, as mentioned before, this choice can influence the probability of a node not get updated in a *pulse*. There are three anti-entropy algorithms:

- Push: An agent will send its states for another to be updated;
- Pull: An agent will ask for another agent's state to update itself;
- Push-Pull (*Sync*): It is a full synchronization between two agents, and both will be updated.

In the Push approach, only the target agent will be updated, after receiving the agent's state. This strategy is useful when there are few nodes on the system carrying that information. When more and more nodes are getting updated, the probability of choosing a node with the same state will increase fast and it will stop performing the operation.

In the Pull approach, only the agent that triggered the operation is updated, after asking to the target node for its state. This strategy is useful when are many nodes on the system carrying that information. Because, when there are few nodes, the probability of contacting some node with the same state is very high.

Finally, the Push-Pull is recommended in the two scenarios described above. It synchronizes the two nodes on the operation, updating both states. This strategy will be used on AngraDB's implementation of this protocol.

After deciding the anti-entropy algorithm, we have to decide when and what will trigger the Gossip process, causing the synchronization between the nodes.

3.2.3 Triggering *Sync*

After explaining all aspects of a Gossip Protocol's *sync* process, it is necessary to understand in what moment this operation occurs. The AngraDB was designed to use two possible strategies of triggering the synchronization in the Gossip Protocol. One is by execution, by one process that wants to force the synchronization to happen, so it calls the method responsible for doing that directly. This can be used when nodes are entering on a cluster and want to synchronize with some other member. Another strategy is by periodic triggering, which chooses a period of time to perform a *sync* process. This strategy is commonly used because it will update the cluster periodically, not worrying about this on the rest of the code. In the case, the configured period of time is too small, the system

can overload with too many requests. That is why this is an important configuration AngraDB to not overload the system.

3.2.4 Usage

This protocol is commonly used to disseminate smaller pieces of information. Usually, it is only necessary to store the metadata about the cluster. At this work, this protocol is used to disseminate the list of *ring_id*s, in order to the nodes on the system to have this list. Also, the "Gossip" protocol is used to warn the nodes if an error has occurred in the connection of two nodes, which is detected by the Hinted Handoff strategy. This is important to form a quorum in order to decide if a node is down or just inaccessible to the node that contacted it in the first place.

3.3 Sloppy Quorum and Hinted Handoff

In order to detect errors in the cluster, to recover from them, and to continue operating in a decentralized way, the failure detection must be done by each node. And for that, the Sloppy Quorum and Hinted Handoff strategies are essential. As a distributed system, every operation will be executed in more than one node, returning multiple responses. To validate the operation as a whole, it is necessary to set some quorum requirements. It is important to set different quorums, for reading and writing operations. On Dynamo paper [7], a strategy for operation validation called Sloppy Quorum is presented. The "W" and "R" configuration are preset for the database to do the quorum validation. This is necessary to adjust the availability or consistency guarantees for these operations. In the case of a highly available system, these quorum has to be set by noticing the system's error rate, in a way to always provide a response to the user. But, to provide consistency for the system, the quorum has to be higher to guarantee that property.

When an operation has some failed cases, this information is used on the strategy called Hinted Handoff, each uses the operation stats to have a hint about what nodes are down or inaccessible. This is a lighter way to discover failures in the cluster because it only will be triggered when an operation is performed.

As mentioned before, the node, which detected the error, uses the Gossip Protocol is used to warn the remaining nodes that a failure occurred, and the failed node is going to be unavailable at least temporarily. Those nodes are going to test the connection with the one with an error. In the case of more nodes could not connect with that node, a quorum would be formed to decide, to remove that node of the cluster metadata, updating the removing the *ring_id* related to the crashed node.

In order to define that it is not a temporary issue, it is possible to configure a timeout for the nodes to wait for a successful connection, in a way to give some time the node with an error to fix itself.

3.4 Vector Clocks

With the occurrence of errors and node crashes, the eventual consistency can be handled to provide to the user the possibility to handle that problem. A key algorithm to solve this problem is called “vector clock”. And as mentioned before, on Table 2.1, this problem helps to provide high availability for write operations, because it frees the system to perform write operation even while the cluster is recovering from errors, generating inconsistency on the cluster. In order to have improved performance, the reconciliation starts only in the next read operation.

In this algorithm, a time reference is generated to be used to identify the version of the data. As the system uses the consistent hash algorithm, every data will be replicated in more than one node. In a successful scenario, all nodes will have the same timestamp reference of each data. However, in the case of errors, those references would become desynchronized. In order to resolve these conflicts, it necessary to set a resolution strategy.

Conflict resolution is a very hard task. It is possible to use the vector clocks and Merkle tree as parameters, the newest data would always be returned for the user. This approach is bad, because not every time, the newest information will be the correct one for the database’s user. A correct way to reconcile these versions is passing the responsibility to who has created that data in the first place: the user. In the case of conflict, all versions will be returned, and the next write operation will resolve the problem, the written data is considered the newest and right one.

Chapter 4

Implementation

This chapter presents the implementation aspects of this project. Some of the algorithms showed on the last chapter have not been implemented because of this project have the focus on the key sharding and the gossip protocol algorithms. All implementations were made using Erlang/OTP. This is a concurrent and functional programming language that runs upon a virtual machine called BEAM.

4.1 Erlang

One of the main Erlang's designers, Joe Armstrong, had shown in his thesis [19] that large systems will probably be delivered containing a considerable number of errors in the software. In the case of a distributed application, the errors will occur with much more frequency. To provide reliability expecting to happen errors, it is necessary to handle these errors in a way to not let the whole system crashes. With this concern, Erlang was built upon a concurrent model, called actor model, responsible for handling errors on the system.

4.1.1 Actor Model

In the actor model, processes are considered actors, being isolated and having one encapsulated job to perform [20]. The communication between actors is mainly made by messages exchange. Actors are categorized into two kinds, supervisors or workers. Workers have the single responsibility of performing the task is assigned to them, they are responsible for doing all kinds of service provided by the system. Supervisors have the single responsibility to take care of the other processes—including other supervisors—and are configured with predefined strategies of handle errors on its children when they happen.

Erlang is built upon a philosophy called "Let it crash!". Its principles are about not trying to prevent all kinds of errors because it will eventually happen. It's easier to let the errors happen and try to handle it or by restarting the process. This job can be done by supervisors, that is why Open Telecom Platform (OTP) provides the implementation of a generic supervisor that supports developers to design its own kind of supervisors and supervision trees.

4.1.2 Distribution

This language also provides great support for distribution. It encapsulates the process communication by using only process's ids on the Erlang's VM to communicate, even if this process is in a different node. This makes the communication between nodes is as simpler as doing between processes in the same computer. This feature is commonly known as "location transparency", and it was very useful on the implementation of this project.

When an Erlang VM is running on distributed mode, a process is executed in background, this process is called Erlang Port Mapper Daemon (EPMD). It is responsible for mapping names, used to identify a node on the systems to ports. It is responsible for doing all the network's hard-work, such as finding other nodes, contacting them. This process is used with the "location transparency" to provide a lighter syntax for node communication in Erlang.

All messages are exchanged inside the Erlang's Distribution Protocol. These are simply defined as the following four steps [21]:

- Low-level socket connection
- Handshake, interchanging known node names
- Authentication

It uses the TCP protocol, by default, to communicate in the network and uses a value called "magical cookie" to encrypt the messages when contacting another node and it also uses that value to authentication, the two nodes, that are contacting with each other, must have the same "magical cookie". For purposes of this project, AngraDB will run in a controlled environment with a local network, there will be no problem to continue using it. But this protocol has to change to provide security when AngraDB start to run in an insecure network, such as the open internet, once that it is necessary to assume this problem in a real-world distributed application. In order to do that, it is common to use TLS protocol, that provides stronger security while exchanging messages in the cluster.

4.1.3 Behaviors

One of the Erlang's main concept, called behavior, was essential to the AngraDB's flexibility on configuration and strategy choice. This concept is used when an application has procedures with very similar responsibilities, but behaves slightly different, and wants to generalize this behavior, providing easiness on switching implementations of the same behavior without having to perform additional changes on the code. In an object-oriented world, an interface would be equivalent to this concept on Erlang. In the case of Erlang, the implementations are modules that present the required callbacks instead of a class that implements the methods specified on the interface.

Every one of the cited Erlang features was closely studied and used to design an distributed architecture. In the following section, this architecture will be described as a set of modules, where every module will have unique responsibilities, following one of the main concepts on Erlang, the actor model.

4.2 Application Architecture

Following the actor model and the "Let it crash" philosophy, the designed architecture for AngraDB in this project, prioritizing the key sharding and the gossip protocol usage, was divided in the following main group modules, shown in Figure 4.1.

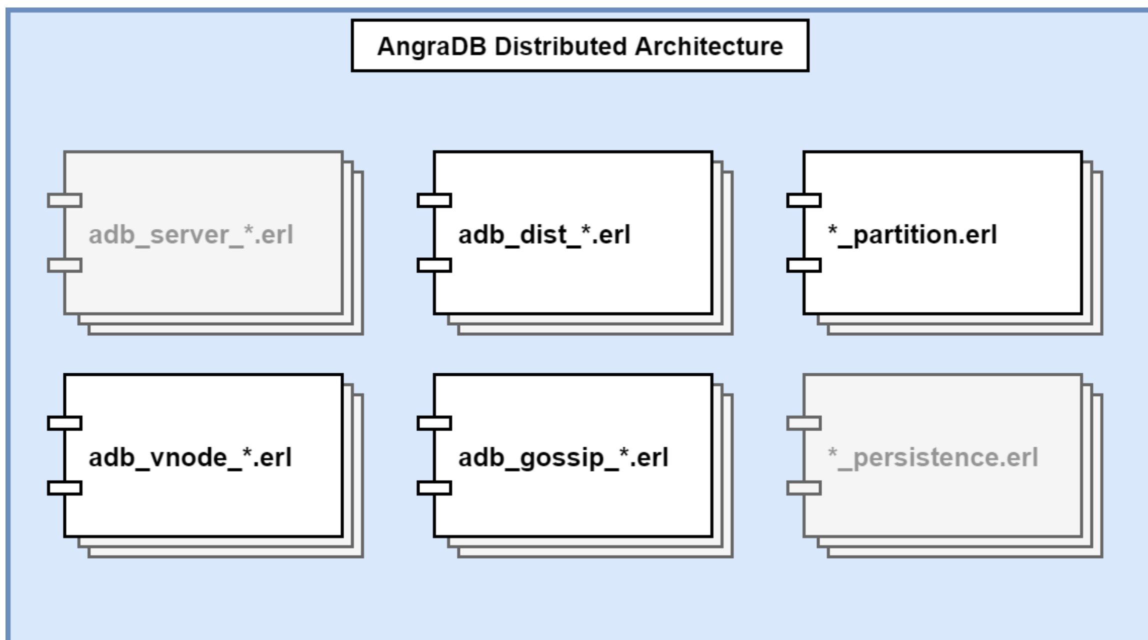

Figure 4.1: AngraDB Architecture Diagram.

In Figure 4.1, the modules that are not blurred represents the modules implemented on this project. However, the other ones are important modules to the AngraDB's operation and are relevant to this project.

4.2.1 The *adb_gossip_** Modules

The *adb_gossip_** modules are responsible for performing the dissemination of data across the cluster using the implemented gossip protocol. Every predefined period of time, this module performs a *sync* operation with a side node. This sync operation compares their nodes states and provides the changes respected to each node on this interaction. In this project, these modules are only used to spread the consistent hash's id of each virtual node to maximum nodes possible. This information is very useful because it lets every node to act as a server, which is a future goal for AngraDB. Thus, it is important for the generation of an id to be assigned for a new node because it depends on the previous one.

It is possible to see in Listing 4.1 a piece of code that captures the essence of the Gossip Protocol. First, we choose a node randomly to synchronize. And then, it triggers its gossip server to start a synchronization. This server contacts another node's gossip server using message exchange. And the contacted node compares their states and returns a message whether it was updated or not. If it was, the node that contacted it will continue to synchronize other nodes. Otherwise, it stops the operation.

```
1 iterate_sync([])     -> ok;
2 iterate_sync(Nodes) ->
3     TNode = adb_utils:choose_randomly(Nodes),
4     Rest = [X || X <- Nodes, X =/= TNode],
5     case gen_server:call(?SERVER, {sync, TNode}) of
6         {ok, synced} when Rest =/= [] -> iterate_sync(Rest);
7         {error, _}   when Rest =/= [] -> iterate_sync(Rest);
8         _Result                       -> {ok, synced}
9     end.
```

Listing 4.1: Gossip's *Sync* operation

4.2.2 The **_partition* Modules

The **_partition* modules are implementations of an Erlang's behavior that is responsible for performing the key sharding. In AngraDB, it is possible to choose between two strategies: Full Partition and Consistent Partition.

The first strategy is used when its necessary that all nodes on the cluster to have every document on the system, this proved to be very useful in scenarios of processing data through work division, because it will improve availability, but it will be harder to let every node get updated as the cluster size increases, because there will be more nodes

to receive that new data. The elasticity brought by this strategy is ideal because the system will behave in the same way, regardless of the fact of nodes entering and leaving all the time. In case of a new node enters in the cluster, the synchronization can be made with any node of the cluster. In this strategy, vector clocks and reconciliation strategies are highly necessary for the system to handle eventual consistency, because, in face of errors, the possibility of desynchronization between nodes will be higher than the second strategy.

```
1  save(Database, Key, {Size, Doc}) ->
2      Targets = [node()|adb_utils:valid_nodes()],
3      Request = {process_request,
4          {all, {save_key, Database, {Key, Size, Doc}}}},
5      ResponseStats = gen_partition:multi_call(Targets, adb_vnode_server, Request),
6      case gen_partition:validate_response(ResponseStats, length(Targets), read) of
7          {success, Response} -> {ok, Response};
8          {failed, Response}  -> {error, Response};
9          {error, Reason}     -> {error, Reason}
10     end.
```

Listing 4.2: Save operation at full partition strategy

Applications that prioritize accessibility instead of getting the newest data. The second strategy is about the already explained “Consistent Hash”, it used on scenarios where is important to have the database highly available, but still having the maximum consistency possible, without losing availability rate. The trade-off in this strategy is more visible on the database configuration, it is possible to tune the number of vnodes, the number of replication copies, and the quorum parameters. These settings are very important for the database behavior in the presence of errors.

```
1  save(Database, Key, {Size, Doc}) ->
2      HashFunc = get_hash_func(),
3      {ok, {_HashKey, Target, VNode}} = map_key(HashFunc, Key),
4      Request = {process_request,
5          {VNode, {save_key, Database, {Key, Size, Doc}}, replicate}},
6      gen_server:call({adb_vnode_server, Target}, Request).
```

Listing 4.3: Save operation at persistent partition strategy

In Listings 4.2 and 4.3, it possible to observe the different behaviors on these strategies. On every strategy, there will be a logic to forward the request for a target node, or nodes in the case of the full partition. In this step, the target process will be chosen and the request will be forwarded to the virtual node manager, the *adb_vnode_server* module. This module will be responsible for executing the request on the chosen virtual node. In Listing 4.3, the method *map_key()* is used to perform the key mapping according to the rules of the Consistent Hash. This mapping strategy can be seen in Listings 4.4 and 4.5, where it uses the methods *find_target_vnode()* and *find_target()* to do the key sharding.

```erlang
find_target_vnode(SpaceSize, HashKey) when is_integer(SpaceSize) ->
    {ok, VNodes} = adb_dist_store:get_config(vnodes),
    PartSize = SpaceSize / VNodes,
    FilterPred = fun(X) -> HashKey =< trunc(PartSize * X) end,
    VNodeId = case lists:filter(FilterPred, lists:seq(1, VNodes)) of
        []      -> VNodes;
        [Id|_] -> Id
    end,
    {ok, {PartSize, VNodeId}}.
```

Listing 4.4: Mapping the key into a virtual node

```erlang
find_target(PartSize, TargetVNode, HashKey) ->
    {ok, RingInfo} = adb_dist_store:get_ring_info(),
    {ok, SortedRingInfo} = adb_utils:sort_ring_info(RingInfo),
    StartPoint = PartSize * (TargetVNode - 1),
    FilterPred = fun({_, {Num, Den}}) ->
        HashKey =< trunc(StartPoint + (Num * PartSize) / Den)
    end,
    Target = case lists:filter(FilterPred, SortedRingInfo) of
        []             -> {Tgt, _} = lists:last(SortedRingInfo),
                          Tgt;
        [{Tgt, _}|_] -> Tgt
    end,
    {ok, Target}.
```

Listing 4.5: Mapping the key into a physical node

First, each key is mapped according to the virtual node, and then, it is mapped according to the physical node. In this way, the cluster is divided into the number of virtual nodes, and each division is also divided by the Consistent Hash's *ring_id*s.

4.2.3 The *adb_vnode_** Modules

The *adb_vnode_** modules are responsible for forwarding the request to the chosen virtual node for the operation. On the database initialization, it is created a number of processes, called virtual nodes, that will behave as a physical node on the cluster, doing documents operations. To guarantee flexibility to the system, even if the system does have just one node, each virtual node will be responsible for a range of keys, even if this range is the maximum possible, containing every key on the system, in the case of the full partition strategy.

Every request that passes to this module is forwarded the desired virtual node to be performed. In the case of full partition, it will forward the request to every virtual node process to be executed. Thus, it has the responsibility to collect the responses and compute the statistics to validate the whole operation as a success or a failed one.

Each virtual node will be responsible for one copy of the database and will perform the database's operations without worrying about conflicts on write operations because each process will have a unique database to write. The conflicts resolution will be handled

on reading operations in a vector clock algorithm. This is important to the virtual node to act as a real physical node, where it will have its own independent database instead of sharing with another node. On every request, a virtual node will use on the **_persistence* module to operate with its database.

4.2.4 The *adb_dist_** Modules

Every request passes to the *adb_dist_server* first, before being assigned to a target node. These modules will choose one of the **_partition* modules to forward the received request to be direct to the right target. This choice is based on the configurable sharding strategy that was chosen at the system initialization.

Besides the forwarding messages duty, *the adb_dist_** modules have the responsibility of storing the global configuration related to distribution and provides it to other processes. This information is exclusively set by the first node to be initialized, which will replicate it to every node that enters on the cluster. All initial node interactions are made on these modules. When a node wants to enter in the cluster, it has to contact some ring member and the configuration are replicated to the newer node. In the practice, nodes are updated with the inherited configuration from the oldest node on the system.

4.2.5 The *adb_server_** Modules

The remaining modules are directly related to the request processing in a distributed way. The *adb_server_** modules have to open a connection to the client and receive all its requests. After receiving, it has to parser request to match with one valid database operation. After this, the processed request is passed to the *adb_dist_server* to be forward to the target node according to the chosen partition strategy.

4.2.6 The **_persistence* Modules

The **_persistence* modules are responsible for the chosen implementation of the AngraDB's persistence behavior. They have to provide the basic operations for handling documents in the database, such as write and read operations. Each implementation is closely related to the chosen storage strategy. On AngraDB, there are the following three:

- HanoiDB: An open-source key-value storage system.
- ETS: A built-in key-value storage engine that uses that will store the documents on the memory.
- Angra B-tree: An implemented B-tree storage engine, implemented as one of the previous goals of AngraDB. Also, it provides great indexing support.

In this project, the only strategy used was Angra B-tree, because of the possibility of customization according to the project's needs and the compatibility of being a project that grew up with AngraDB. Also, this persistent strategy has the capability of doing indexing and performing fast searches.

4.2.7 Design for Configuration

As a core concept, the AngraDB's flexibility in configuration has been taken care as one of its most important features. In a distributed system there are choices to make that will determine the usability of the database in a specific scenario.

As a distributed system, all nodes are identified by some name. By default, a name is generated by the user in Erlang. To make easier to initialize, AngraDB generates randomly these names. Its possible to define patterns to that names on configuration.

When configured to do so, AngraDB uses consistency hash as a sharding strategy. In that scenario, it's possible to set the desired hash function and the "N", "W", "R" parameters, inherited from DynamoDB [7]. These parameters represent the number of replications for each operation, and the write and read quorums, respectively.

To use Gossip Protocol, it is possible to define a synchronization interval. This will be used to define the frequency each node will contact another to *sync*.

4.3 Cluster Architecture

In a way to implement a reliable system, the design choices were always focused on the simplest option. The system has to have the flexibility to be replicated into a cluster in a way that it is easier to be deployed.

As mentioned before, the Erlang's runtime spawns EPMD to care about the portmapper and node discovery. However, that process depends on a shared network to find other nodes in the cluster. So, in production, AngraDB has to be deployed in a cluster, where the nodes belong to the same local network. In order to configure an cluster in a global network, it is necessary to re-implement the Erlang's native support for distribution. This capability have not been implemented on this work.

Besides that, to deploy the application, it is important to care about the ports used by the application. By default, EPMD uses the port 4369 to connect to other nodes, but, after connecting to them, it opens a new port for every running application. In a production environment, it is necessary to define the port range required by the application. However, only the newest versions of Erlang let us choose the port range for this connection. In older versions, the ports were being opened on background indefinitely.

4.3.1 Membership

In a cluster with multiple nodes, there must be the possibility to the newest node to become a member of the cluster in an easier and safer way. There were two approaches to resolve this problem: a centralized way, by using a detached manager process to handle membership of the cluster or a decentralized, where the new node can connect with any node to enter into the cluster, and for this end, each node has the capability to manage the entering of this node.

One of the goals of this project is to eliminate the single points of failure, the decentralized option had shown to be the right choice to serve the perform a membership process in a reliable way. However, as mentioned before, the support to handoff operations would not be implemented, this support is essential to the node, which had been contacted by the entering node, to compute and to set instructions to other nodes to transfer their documents to this new node according to the chosen sharding strategy.

4.3.2 The ***ring_id***'s Assignment Strategy

When designing the consistent hash for the system, it was necessary to choose some strategy to assign *ring_id*s to nodes entering in the cluster. In order to provide more load balancing for the documents, the ranges of hash values have to be equally separated in a way to receive the same range of documents. But in a scenario where there are nodes entering or leaving the cluster, this assignment has to be incremental, in a way that its necessary only the previous *ring_id*s are necessary to compute a new one.

The chosen strategy was to always divide the whole cluster into 2^n ranges, regardless of the number of nodes on the cluster. It takes advantage of the fact of the corresponding values between fractions of different factors, in special the 2^n factor.

When we divide some value in half, we have two fractions, corresponding to the $\frac{1}{2}$ fraction of that value. However, in a consistent hash, to divide the whole space produced by a hash function in half, it has to assign two *ring_id*s, one corresponding to the $\frac{2}{2}$ fraction of the whole space and another corresponding to $\frac{1}{2}$. When we passed to divide the cluster into four ranges, there some corresponding values to the older *ring_id*s. The $\frac{2}{2}$ fraction has the same value compared to a $\frac{4}{4}$ fraction, and in the same way, the $\frac{1}{2}$ are equal to $\frac{2}{4}$. The new *ring_id*s will have odd values in the upper-side of the fraction ($\frac{1}{4}$ and $\frac{3}{4}$). When dividing by the next 2^n, the same result will appear. The odd value of a numerator will always have an even value with we double the value of the factor.

The explanation above is a little basic, but it a good way to show how the *ring_id* assignment strategy works. The initial node always receives the $\frac{1}{1}$ value as its *ring_id*. And so one, every that enter in the cluster, it receives next odd numerator, except for the

Table 4.1: Ring configuration according to the number of nodes on the cluster.

Cluster Size	**Nodes (*ring_ids*)**							
	1st	**2nd**	**3rd**	**4th**	**5th**	**6th**	**7th**	**8th**
1	A ($\frac{1}{1}$)	none	none	none	none	none	none	none
2	B ($\frac{1}{2}$)	A ($\frac{2}{2}$)	none	none	none	none	none	none
3	C ($\frac{1}{4}$)	B ($\frac{2}{4}$)	A ($\frac{4}{4}$)	none	none	none	none	none
4	C ($\frac{1}{4}$)	B ($\frac{2}{4}$)	D ($\frac{3}{4}$)	A ($\frac{4}{4}$)	none	none	none	none
5	D ($\frac{1}{8}$)	C ($\frac{2}{8}$)	B ($\frac{4}{8}$)	D ($\frac{6}{8}$)	A ($\frac{8}{8}$)	none	none	none
6	D ($\frac{1}{8}$)	C ($\frac{2}{8}$)	E ($\frac{3}{8}$)	B ($\frac{4}{8}$)	D ($\frac{6}{8}$)	A ($\frac{8}{8}$)	none	none
7	D ($\frac{1}{8}$)	C ($\frac{2}{8}$)	E ($\frac{3}{8}$)	B ($\frac{4}{8}$)	F ($\frac{5}{8}$)	D ($\frac{6}{8}$)	A ($\frac{8}{8}$)	none
8	D ($\frac{1}{8}$)	C ($\frac{2}{8}$)	E ($\frac{3}{8}$)	B ($\frac{4}{8}$)	F ($\frac{5}{8}$)	D ($\frac{6}{8}$)	G ($\frac{7}{8}$)	A ($\frac{8}{8}$)

case this value is bigger than the factor of the fraction. In that case, the factor doubles and the numerator return to the value one. The assignment order will be $\frac{1}{1}$, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{3}{4}$, $\frac{1}{8}$ and so on. In this way, we can guarantee that the cluster will be equally divided, at least when there are 2^n nodes on the cluster. And we have an incremental assignment algorithm because it is only necessary the previously generated *ring_id* in order to get a new one.

It is shown in Table 4.1, the configuration of ring architecture, based on this strategy, and according to how many nodes have entered on the cluster, in the case 1-8 nodes. On the table, the letter is referred to what order that node has entered on the cluster. In Listing 4.6, it is possible to see this assignment strategy in the implementation written in Erlang.

```
1 generate_new_ring_id(LastRingId) ->
2     {Num, Den} = LastRingId,
3     if
4         Num + 2 > Den -> {1, Den * 2};
5         true          -> {Num + 2, Den}
6     end.
```

Listing 4.6: The generation of *ring_id*s

4.3.3 Highly Available Server

As one of the main problems cited by this project, a reliable application has to handle the death of one of its nodes. We have been dealing with this by using a Consistent Hash and the Gossip Protocol algorithms. However, those approaches only are necessary

for providing reliability and error tolerance for AngraDB's file system. But in terms of providing high availability to the server, in a way to handle multiple requests without letting the user without any connection access.

The chosen strategy here was to have one server per virtual node, and according to this AngraDB's configuration, it is possible to have multiple servers running in each physical node in the cluster. This choice has the advantage of providing to the user a wider range of servers to communicate, not being at the mercy of the errors of a single node.

In order to make this feature possible, it was necessary to design the system in a way that it is so generic that every node has the resources to do the consistent hash's mapping. For that purpose, nodes share the list of *ring_id* between each other using the Gossip Protocol to synchronize that information across the cluster.

This feature also is important for the future versions of AngraDB, such as a load balancing to choose the nearest node to connect the user and its server in a way to reduce geographic latency on each request, and that can be maximized when the database could be installed in remote hosts around the world.

Chapter 5

Experiments

This chapter shows the results from the attempt to validating the most important algorithms on this project. The development of the distributed architecture was put on the test. To prove the features specified in the other chapters, some kinds of tests have to be defined.

Following the main goals of this project, which were to implement the distributed algorithms responsible for providing an initial design to AngraDB to operate in multiple hosts: Consistent Hash and Gossip Protocol. All defined tests cases were defined in a way to test the reliability of the implemented code.

5.1 Environment

In way of simulating a production environment, the first attempt was to start instances of AngraDB in multiple physical hosts. Besides the fact that this strategy seemed to be easier, the difficulties to configure the environment to use Erlang's runtime on each host and have let the testing strategy to be changed.

The second approach was to actually use virtual machines in a cloud platform. The chosen service was Azure, which provides good support for deployment and has some facilities for using VMs for personal usage. However, there were too many configurations to run Erlang and AngraDB on those machines, sometimes it was necessary to configure parameters that will never be used on the project, even less on testing. These problems had made the approach to focus more on the required environment for testing and deployment Erlang applications.

After studying about Erlang's deployments, we let to the third approach that uses Erlang's releases feature, encapsulates the environment inside an executable ready to be distributed. After that, thinking about to reduce the cost of configuration and to enable the testing to be local even simulating distribution without overload the host, in the case of

using multiple VMs running simultaneously, we have chosen a new approach. It is a lighter virtualization approach, based on containers, which has better performance compared to the virtualization based on hypervisors [22]. Recently, this kind of virtualization has been popularized by Docker, in order to make the deployment process easier.

One of the main reasons why local testing was chosen instead of a fully integrated cloud platform, which has many features that can help AngraDB to improve its reliability in a production environment, was that, for now, it is not necessary to have this kind of platform to implement tests for the features implemented on this project.

5.2 Local Distribution with Docker Compose

As mentioned before, containers have better performance than hypervisor-based virtual machines. When initializing container, virtualization happens in the operating system level, it does not try to create virtual hardware to support each virtual machine, as the hypervisor approach does, it uses one virtual layer to provide to all containers the same hardware. This improves the portability of the container's image and its efficiency, where a container image takes much less space than a hypervisor-based image [23]. A comparative illustration is shown in Figure 5.1.

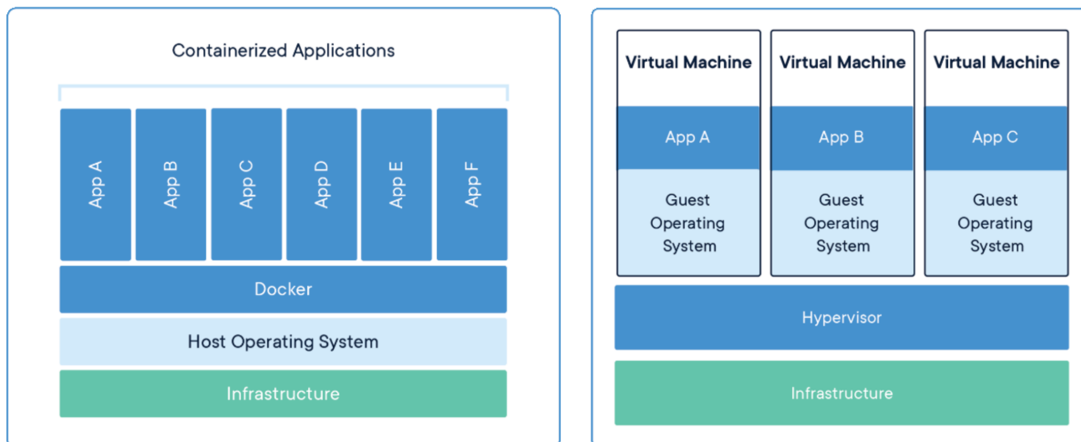

Figure 5.1: Hypervisor-based and Container-based Virtualizations (Source: [23]).

After configuring the image building script, where the AngraDB's release is made. The deployment of AngraDB has become easier than ever. And with help of the Docker Compose utility, it had made the cluster deployment very easy. This tool is distributed with Docker Engine (responsible for running the containers), and it is used to define the set of containers that will be run by Engine. Thus, it provides the option to define a local

network between the containers that will be executed. This feature is essential to build a cluster using Erlang.

5.3 Testing Consistent Hash

As one of the main goals when designing the distribution architecture for AngraDB, the distribution of documents through the cluster provides the capability of receiving a great number of documents without overloading some node and distributing the documents equally. These tests were made in order to prove the load distribution provided by the Consistent Hash and the *ring_id* assignment strategy. Those have been performed by saving a considerable amount of documents on the cluster and verifying the distribution of the saved documents through the cluster. The high availability of the server is also tested here, by performing the request on every available server. These tests were made by send requests to all servers on the cluster, where there were nine servers in the case of three nodes and twelve servers in the case of four nodes. The average was computed and plotted at the presented histograms.

For these tests, we have disregarded the documents that are copied on Consistent Hash's replication strategy, in a way to only count the documents that are directly mapped to that node. And as default, AngraDB was configured to use three vnodes per physical node. And the database was configured to use MD5 function to map the keys and it was saved one thousand of small documents ($\sim$ 20 bytes). Those documents have always the same format: A simple JSON {"*foo*" : "*bar*"}. These tests were made by using a desktop application [24], written in Node.js, that communicates with AngraDB through a TCP connection. In this application, it was possible to tune the number of times that the application will perform a "save" operation. The results of those tests were collected exclusively from logging files that AngraDB had produced.

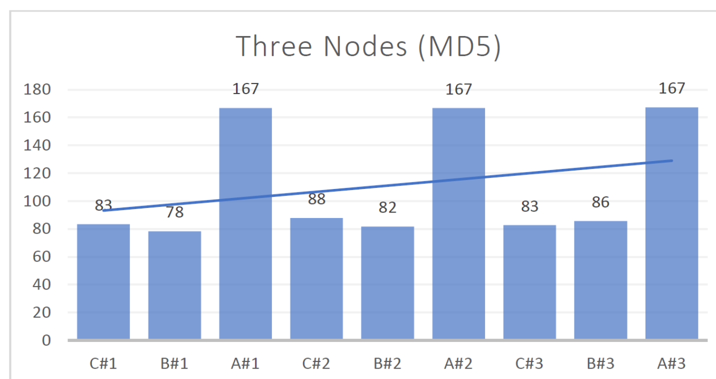

Figure 5.2: Load distribution when using MD5 in a three-nodecluster.

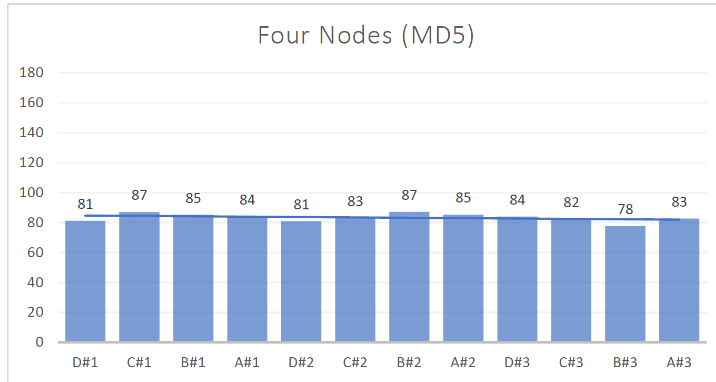

Figure 5.3: Load distribution when using MD5 in a four-node cluster.

On Figures 5.2 to 5.3, there are histograms that show the load distribution in a cluster of three and four nodes. In Figure 5.2, it is visible the difference of the amount of document stored on the vnodes of the physical node A. This is behavior is expected because the cluster does not have a size of 2^n as mentioned before, Instead, node A has $\sim 50\%$ of the documents, and nodes B and C have $\sim 25\%$ each. This behavior is expected because the *ring_id* assignment for these number of nodes (shown in Table 4.1) provides this configuration. However, in Figure 5.3, it is explicit that the load was divided into approximately equal parts.

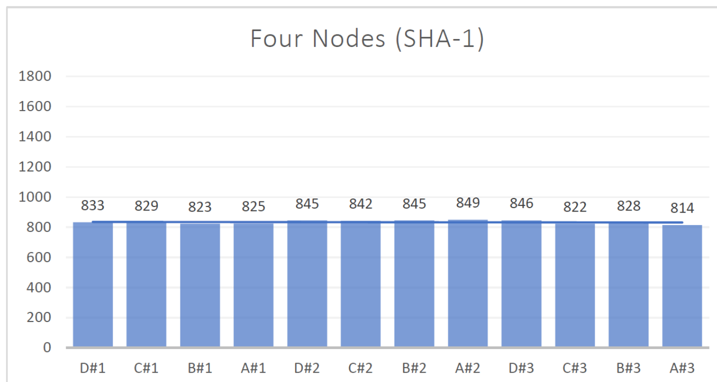

Figure 5.4: Load distribution when using SHA-1 in a four-node cluster.

After testing a load distribution using MD5 to map the keys on the cluster, it was necessary to test using another hash functions (SHA-1, SHA-256, SHA-384, and SHA-512) in order to see the behavior of AngraDB when using these functions. In order to simulate in a possible environment, where these functions are necessary because of their wider hash space, we used a bigger load for these tests, so it was stored ten thousand documents per test with the same format we see on the previous tests with MD5 hash

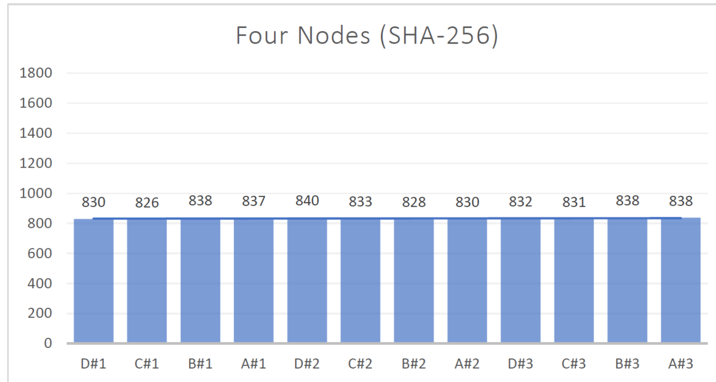

Figure 5.5: Load distribution when using SHA-256 in a four-node cluster.

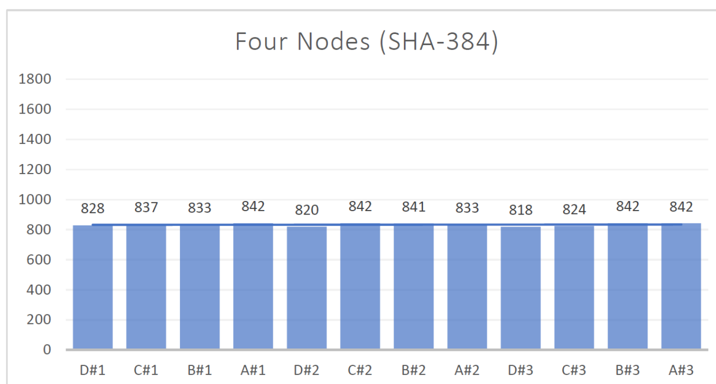

Figure 5.6: Load distribution when using SHA-384 in a four-node cluster.

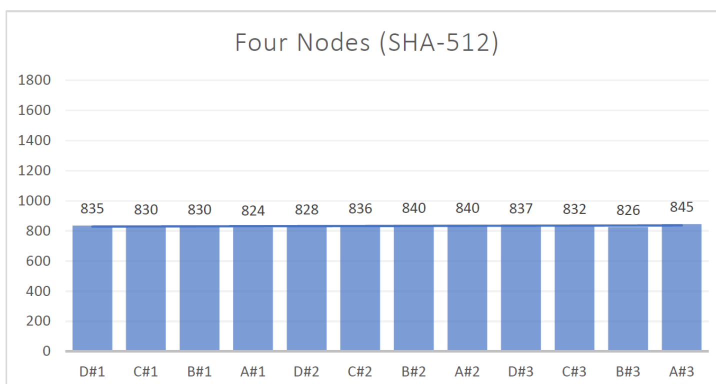

Figure 5.7: Load distribution when using SHA-512 in a four-node cluster.

function. On Figures 5.4 to 5.7, it is shown histograms referred to the tests using the hash functions. It is possible to observe that changing the available hash function will not have an influence on the load distribution of the cluster.

5.4 Testing Gossip Protocol

The Gossip Protocol is used for synchronizing state between nodes in a distributed architecture, and, as mentioned before, this protocol has the benefit of not relying on the fact that a node has access to all nodes in the cluster, it performs P2P synchronizations. This is important for the system to continue to provide the CAP's partition tolerance guarantee. The main goal of these tests was to prove the capability to update the cluster using Gossip Protocol. The strategy for these tests was simple. First, a cluster with an arbitrary number of nodes is created by using the Docker Compose script [25]. Second, it collects data from AngraDB's logging files every time a Gossip Protocol's *pulse* happens.

When testing in a small cluster, running locally, it is visible that all nodes are updated after the first *pulse*. Thus, we have verified that even in a cluster with 16 nodes (48 virtual nodes), the system continues to synchronize on the first or second *pulse*, however, it was not possible to see the real use for this protocol, which is in the presence of the network partition on the cluster. This can be explained by the fact of the tests are being executed in a controlled environment, which is very difficult to present a connection problem.

For the fact that the algorithm has shown results that it can synchronize the cluster, it can be stated that this protocol works for its main purposes. However, additional tests are required, those that simulate bad environments where the protocol has to handle and is capable to work normally.

Chapter 6

Conclusion

In this work, we designed and implemented a distributed architecture on top of the non-relational database AngraDB. This implementation brings several benefits, including load distribution, high availability, partition tolerance, and elasticity to handle the entering and leaving of nodes. It also allowed the implementation of a MapReduce module [26] for AngraDB, as a result of another final project. The architecture is based on the Consistent Hash algorithm which requires a Ring architecture.

6.1 Learned Lessons

It is fear to say that some of the most important lessons learned on this project were from the Erlang's knowledge. Many aspects of functional programming and concurrency have been deeply comprehended, such as the pattern matching philosophy and the actor model.

Also, while testing the distributed architecture for AngraDB, we have learned an important lesson: it is difficult to test distributed architecture, we must have the resources necessary to simulate an environment with multiple hosts. Also, it is even more difficult to test the auto-healing capabilities of the system, we had to simulate hard situations to cause the system to heal itself. The problems we had while testing AngraDB had led to the search for testing strategies specially designed to a distributed system.

6.2 Future Works

The next steps for this project are to continue to test this project in other contexts that need a distributed architecture in order to fix bugs and improve the database reliability. The main goal here is to always improve the features implemented on this project.

Some aspects of failure detection and recovery have not been implemented on this project. Between them, are the Hinted Handoff and the Gossip Protocol use for these purposes. These features are necessary to the AngraDB's capacity of self-healing in the case of errors. Also, one of the most important features to be implemented next on AngraDB is vector clocks and Merkle three in order to improve the database's availability on writing operations and to recover from consistency issues.

Thus, the Handoff operations should be implemented on this system, in order to guarantee that a new node will always be updated when it enters on the cluster. These operations will have an impact on the database performance, so it should operate in background.

As mentioned before, we have been searching for more reliable testing strategies for distributed systems, we have noticed a testing strategy designed especially for self-healing distributed application. It was called "Chaos Engineering", a strategy used to simulate a cluster environment where errors occur frequently and randomly [27]. This strategy could be useful use test AngraDB's newest architecture and its self-healing capacity. Another strategy that was considered on the start of this work is to use formal methods (using PRISM model checker) in order to provide reliability to the design of the project, not only the implementation.

References

[1] *Nosql databases.* `http://nosql-database.org`, visited on 2019-01-22. 1

[2] J Han, Haihong E, G Le and J Du: *Survey on nosql database.* 2011 6th International Conference on Pervasive Computing and Applications, pages 363–366, 2011. 1

[3] *Angra's github organization.* `https://github.com/orgs/Angra-DB/dashboard`. 1

[4] M Michael, J Moreira, D Shiloach R W. Wisniewski: *Scale-up x scale-out: A case study using nutch/lucene.* IEEE International Parallel and Distributed Processing Symposium, 2007. 2

[5] D Karger, E Lehman, T Leighton R Panigrahy M Levine D Lewin: *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web.* Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97), pages 654–663, 1997. 3, 12, 13, 15

[6] A Demers, D Greene, C Hauser W Irish J Larson S Shenker H Sturgis D Swinehart D Terry: *Epidemic algorithms for replicated database maintanance.* Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, 1987. 3

[7] G DeCandia, D Hastorun, M Jampani G Kakulapati A Lakshman A Pilchin S Sivasubramanian P Vosshall W Volgels: *Dynamo: Amazon's highly available key-value store.* Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 41(06):205–220, 2007. 3, 4, 7, 8, 14, 15, 17, 20, 29

[8] Brewer, E: *Towards robust distributed systems (abstract).* Proceedings of the nineteenth annual ACM symposium on principles of distributed computing, 2000. 4, 5

[9] Brewer, E: *Cap twelve years later: How the "rules" have changed.* Computer, 45(02):23–29, 2012. 6

[10] V Gudivada, D Rao, V Raghavan: *Renaissance in database management: Navigating the landscape of candidate systems.* Computer, 49(04):31–42, 2016. 6

[11] *Couchdb documentation - 4. replication.* `http://docs.couchdb.org/en/stable/replication/index.html`, visited on 2018-04-21. 8

[12] Klophaus, R: *Riak core: building distributed application without shared state.* ACM SIGPLAN Commercial Users of Functional Programming, 10(14), 2010. 9, 15, 16, 17

[13] A Chaffari, N Chechina, P Trinder J Meredith: *Scalable persistent storage for erlang: Theory and practice*. Proceedings of the twelfth ACM SIGPLAN workshop on Erlang, 12(01):73–74, 2013. 9

[14] *Mongodb architecture guide (white paper)*. `https://www.mongodb.com/collateral/mongodb-architecture-guide`, visited on 2018-04-22. 10, 11

[15] Schneier, B: *Applied Cryptography*. John Wiley & Sons, 1996. 13

[16] A J Menezes, J Katz, P C van Oorschot S A Vanstone by: *Handbook of Applied Cryptography*. CRC Press, 1996. 13

[17] *Secure hash standard*. `https://csrc.nist.gov/csrc/media/publications/fips/180/3/archive/2008-10-31/documents/fips180-3_final.pdf`. 13

[18] A S Tenenbaum, M Van Steen: *Distributed Systems Principles and Paradigms*. Pearson Prentice Hall, 2007. 17, 19

[19] Armstrong, Joe: *Making reliable distributed systems in the presence of software errors*, 2003. 22

[20] Hebert, F: *Learn you some Erlang for Great Good*. No Starch Press, 2013. 22

[21] *Distribution protocol*. `http://erlang.org/doc/apps/erts/erl_dist_protocol.html`, visited on 2018-05-10. 23

[22] R Morabito, J Kjällman, M Komu: *Hypervisors vs. lightweight virtualization: A performance comparison*. 2015 IEEE International Conference on Cloud Engineering, pages 386–393, 2015. 34

[23] *What is a container*. `https://www.docker.com/resources/what-container`, visited on 2019-01-21. 34

[24] *Angradb's node.js desktop client repository*. `https://github.com/Angra-DB/client-electron`. 35

[25] *Docker compose's script for angradb's cluster initialization*. `https://github.com/Angra-DB/core/blob/develop/docker-compose.yml`. 38

[26] Marques, F: *A mapreduce architecture applied to a distributed documental database*. jan 2019. 39

[27] A Basiri, N Behnam, R de Rooij L Hochstein L Kosewki J Reynolds C Rosenthal: *Chaos engineering*. IEEE Software, 33(03):55–41, 2016. 40