An Eventually Perfect Failure Detector in a High-Availability Scenario

Henrique Sousa Pinheiro

Dissertation presented to the School of Technology and Management of Polytechnic Institute of Bragança to obtain the Master Degree in Information Systems. In the scope of double degree with the Federal University of Technology - Paraná.

Work advised by:

Prof. Rui Pedro Lopes

Prof. Rodrigo Campiolo

Bragança

2018–2019

An Eventually Perfect Failure Detector in a High-Availability Scenario

Henrique Sousa Pinheiro

Dissertation presented to the School of Technology and Management of Polytechnic Institute of Bragança to obtain the Master Degree in Information Systems. In the scope of double degree with the Federal University of Technology - Paraná.

Work advised by:

Prof. Rui Pedro Lopes

Prof. Rodrigo Campiolo

Bragança

2018–2019

Dedication

Foremost I dedicate this work to my family Joilma, Carminho and Felipe who always supported me until this day. I also would like to dedicate this work to my friends Andressa, Bruno, Daniel, Jonathan, Juliana, Sávio, and Vitório for their friendship and for always encouraging me to learn new things.

Acknowledgments

The realization of this work counted on a lot of support and incentives that made this possible.

I'm grateful for Professor Rui Pedro Lopes for guiding me through this project and sharing his knowledge, time and passion for this beautiful country. I'm grateful for Professor Rodrigo Campiolo for his great classes, his opinions, and guidance. Finally, I extend my gratefulness for both institutions UTFPR and IPB, for giving me this singular opportunity to learn new things while experiencing a different culture of which I will always care an affection.

Abstract

Modern-day distributed systems have been increasing in complexity and dynamism due to the heterogeneity of the system execution environment, different network technologies, online repairs, frequent updates and upgrades, and the addition or removal of system components. Such complexity has elevated the operational and maintenance costs and triggered efforts to reduce it while improving its reliability.

Availability is the ratio of uptime to total time of a system. A High Available system, or systems with at least 99.999% of Availability, imposes a challenge to maintain such levels of uptime. Prior work shows that by using system state monitoring and fault management with failure detectors it is possible to increase system availability.

The main objective of this work is to develop an Eventually Perfect Failure Detector to improve a database system Availability through fault-tolerance methods. Such a system was developed and tested in a proposed High-Availability database access infrastructure.

Final results have shown that is possible to achieve performance and availability improvements by using, respectively, replication and a failure detector.

Keywords: Distributed systems, failure detection, high availability

Resumo

Os Sistemas distribuídos modernos têm aumentando em dinamismo e complexidade devido à heterogeneidade do ambiente de execução, diferentes tecnologias de rede, manutenção *online*, atualizações frequentes e a adição ou remoção de componentes do sistema. Esta complexidade tem elevado os custos operacionais e de manutenção, incentivando o desenvolvimento de soluções para reduzir a manutenção dos sistemas enquanto melhora sua confiabilidade.

Disponibilidade é a razão do tempo de atividade sobre um intervalo de tempo total. Sistemas de Alta Disponibilidade, ou seja, que possuem pelo menos 99.9999% de Disponibilidade, representam um grande desafio para manter tais níveis de operacionalidade. Trabalhos anteriores mostram que é possível melhorar a Disponibilidade do sistema utilizando o monitoramento de estados do sistema e o gerenciamento de falhas com detectores.

O objetivo principal deste trabalho é desenvolver um Detector de Falhas Eventualmente Perfeito que pode melhorar a Disponibilidade de um sistema de base de dados através de uma arquitetura de Alta Disponibilidade.

Os resultados finais mostram que é possível ter ganhos de desempenho e disponibilidade utilizando, respectivamente, métodos como replicação e detecção de falhas.

Palavras-chave: Sistemas Distribuídos, detecção de falhas, alta disponibilidade

Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface. 10, 20, 30–32, 35

DAG Directed Acyclic Graph. 30

DNS Domain Name System. 18

FIFO First-In-First-Out. 12

IP Internet Protocol. 10, 11, 13, 18, 37

IPC Inter-Process Communication. 12

JDBC Java Database Connectivity. 16

JVM Java Virtual Machine. 30

MoM Message Oriented Middleware. 31

MTBF Mean Time Between Failure. 6

MTTF Mean Time to Failure. 5

MTTR Mean Time to Repair. 6

OLTP Online Transaction Processing. 21

RDBMS Relational Database Management System. 16, 17

RMI Remote Method Invocation. 14

RPC Remote Procedure Call. 14

SQL Structured Query Language. 17, 41

SSL Secure Sockets Layer. 13

TCP Transmission Control Protocol. 10–12

TPCC TPC Benchmark C. 20–22

tpmC Transactions per minute-C. 21, 26

UDP User Datagram Protocol. 11, 12, 31, 37

VM Virtual Machine. 22, 23, 27

WORA Write Once, Run Anywhere. 30

WSREP Write-Set Replication. 20

XML Extensible Markup Language. 22

Chapter 1

Introduction

A distributed system can be viewed as a set of hardware or software components, located at networked computers, that communicate and coordinate their action only by exchanging messages. From this definition one can derive three main characteristics of distributed systems, namely: (i) components concurrency; (ii) lack of global clock; (iii) independent components failures [1].

Availability is defined as the probability that a system is operating satisfactory at any point in time under stated conditions or the ratio of up time to total time [2]. A High Available system is one that can deliver at least 99.999% of Availability, bounded to a maximum of 5 minutes of downtime when considering a whole year of system activity [3].

1.1 Theoretical framework

Modern-day distributed systems have been increasing in complexity and dynamism due to the heterogeneity of the system execution environment, different network technologies, online repairs, frequent updates and upgrades, and the addition or removal of system components. Such complexity has elevated the operational and maintenance costs and triggered efforts to reduce the system maintenance time and costs while improving its reliability, also referred as Availability [2][4].

Prior work was done to increase system Availability through system state monitoring and fault management with failure detectors, applying counter measures that could help guarantee that the system properly provides its services [4].

1.2 Objectives

The main objective of this work is to develop an Eventually Perfect Failure Detector and test it in a High-Available infrastructure. Starting from the main objective the following specific objectives were derived:

- Introduce and set up and high available architecture for database access.
- Explore the architecture in order to verify its performance with different layouts and identify its limitations in terms of failure detection.
- Implement a failure detection module and add it to the architecture in order to mitigate possible unwanted events by using failure detection.

1.3 Document Structure

After the introduction, this document is structured as follows: Chapter 2 introduces the fundamental concepts behind failure detection, types of failures, failure detectors and provides a background overview of the communication methods used in this project; Chapter 3 introduces a high available architecture and discusses its performance and limitation in terms of failure detection; Chapter 4 explores the implementation details of an eventually perfect failure detector and its integration with a database proxy; Chapter 5 presents the final conclusions and future works of this project.

Chapter 2

Context and Concepts

This chapter introduces the fundamental theories behind failure detection by presenting the concept of a fault-tolerant system on Section 2.1, classifying types of failures on Section 2.2 and discussing metrics for how a system operational state can be measured on Section 2.3. On Section 2.4 it is presented the concept of an unreliable failure detector along with its different types, and at Section 2.5 the network communication tools used on this project are discussed.

2.1 Fault-Tolerant Systems

A distributed system can be viewed as a set of hardware or software components, located at networked computers, that communicate and coordinate their action only by exchanging messages [1]. Such components could be subject to individual failures. Thus it is necessary that system designers have ways to express and properly handle such behavior.

A system is said to be *fault-tolerant* when, in case of a component failure, it exhibits a well-defined failure behavior or the system masks component failures to users [5]. Designing fault-tolerant systems can be a difficult task as it is necessary to understand and control all of the individual components aspects when they are properly functioning as well, in some more complex case, when there is a probability of a component failure.

Thus, to design a fault-tolerant system it is necessary to have a good definition of how the system and its components could depend on different components or even on different systems, and to have a model for expressing such dependencies and behavior on its architecture.

A model introduced by Cristian [5] states that systems architectures can be explained in terms of *service*, *server* and a *depends upon* relation. A specification defines what operations a component is able to do and a component that provides a set of operations, i.e. a service, through a given input, is called a *server*. A server u *depends upon* a server r if the correctness of u behavior *depends* on the correctness of r behavior. In this case, the server u is said to be an *user*, (i.e. a client), of r, while r is said to be a *resource* of u. It is important to point out that such *depends upon* relation is not about flow of execution but dependency on the correctness of the server component. This abstraction in terms of services, servers and clients can be extensible to both software and hardware domains [5].

2.2 Failure Classification

A fault (or failure) can be either a hardware defect or a software/programming mistake (bug) whereas an error is a manifestation of the fault/failure/bug. Both failure and error can spread through the system and potentially propagate itself as other components could depend directly, or indirectly, on the output or even on the proper functional state of another component [6].

In order to mitigate and limit error and failure propagation, systems designers incorporate *fault containment zones* into the system. A fault containment zone is a concept by which components prone to error or failure can be isolated in order to prevent direct dependency of another component for proper functioning. Fault containment zones are implemented by using component redundancy and some agreement based on the output of other components [5][6].

Failures can be classified according to several aspects. To start with, they may be hardware or software related. In the context of this work, only software failures are considered.

Regarding their duration, failures can be classified into *permanent*, *transient*, or *intermittent* [6]. A *permanent failure* reflects the permanent absence or malfunctioning of a component. A *transient failure* is one that causes a component to malfunction for some arbitrary period of time and after such period the functionally is restored again. An *intermittent failure* is one that it never goes away but it oscillates on being active, when a component malfunctions, or inactive, when a component is working properly.

Concerning their behavior, failures can be classified as: *timing failure* when, although correct, the output fails to come in the specified time interval; *response failure* when the component output, or its state, is incorrect; *omission failure* when the component somehow omits response to an input; *crash failure* when the component is unable to respond until it is restarted [5].

A failure can also be categorized as *benign* or *malicious*. A *benign* failure causes a component to go dead and, by consequence, could be easier to detect and deal with. A *malicious* (or Byzantine) failure causes a component, although looking functional, to produce erroneous output to the system and propagating the failure [6].

2.3 Reliability, Availability and Serviceability (RAS)

To get a view of the system life-cycle it is necessary to collect data about the system state and to have ways for categorization and comparison. *Availability*, *Reliability* and *Serviceability* are metrics that represent an effort to quantify how a system is expected to function over some period of time.

Reliability, denoted as $R(t)$, is the probability of a system being up, continuously, in the time interval $[0, t]$. This measure is suitable for systems in which even a momentary disruption can prove costly [6]. Related to reliability there are Mean Time to Failure

(MTTF), Mean Time to Repair (MTTR) and Mean Time Between Failure (MTBF), where the latter is derived from MTBF = MTTF + MTTR.

Availability, denoted by $A(t)$ and calculated by A = MTTF ÷ MTBF, is the average fraction, often presented as a percentage, of time over the interval $[0, t]$ that the system is up. This measure is appropriate for applications in which continuous performance is not vital but where it would be expensive to have the system down for a significant amount of time [6].

A **High-Available** system is one that can deliver 99.999% of Availability. As shown on Table 2.1, for a system to be considered High-Available, there is a need for fewer failures and fast repair time as the downtime decreases in orders of magnitude from a class to another [3].

System Type	**Downtime (min/year)**	**Availability**	**Class**
unmanaged	50,000	90%	1
managed	5,000	99%	2
well-managed	500	99,9%	3
fault-tolerant	50	99,99%	4
high-availability	5	99,999%	5
very-high-availability	.5	99,9999%	6
ultra-availability	.05	99,99999%	7

Table 2.1: Availability classes

Serviceability refers to the ease of performing diagnosis and repair of a system. It is also referred to as the maintainability of a system. In a more broader definition, maintainability accounts for the probability of a successful corrective maintenance action within a specified period of time and also taking into consideration all of the technical know-how, and human resources (if necessary), needed in the action as a whole [7].

Systems, or even components, can be composed of multiple elements. Thus it can be hard to calculate the discussed metrics and to define whether a system, or component, has failed when one of its parts has failed or when all of its parts has failed. Some threshold

could be applied but, still, there is not a consensus about how such threshold should be effectively calculated.

2.4 Failure Detectors

As coordination happens by passing messages over a network, and there is no physical global clock, distributed systems have an intrinsic *asynchronous* behavior. Thus, the global state of a distributed computation is derived from the state of all processes and communication channels involved [8].

This asynchronous characteristic, as proved by Fischer, Lynch, and Paterson [9], imposes an impossible solution for consensus and atomic broadcast problems, with the former being a problem where reliable processes over a distributed network must agree on the same value or state, and the latter a problem where all correct processes receive the same set of messages in the same sequence. Such impossibility, in a totally asynchronous model of computation, happens due to the difficulties in determining whether a process has crashed, i.e. is unreliable, or it is only taking a considerable amount of time to respond.

To tackle this problem, Chandra and Toueg [10] augmented the asynchronous model of computation by recognizing its limitations and allowing mistakes to be made by an external failure detector mechanism. Such augmentation introduced the concept of a *unreliable failure detector* model where each process of the distributed system has access to a local *failure detector module*. The *failure detector module* works by maintaining a list of *suspected* crashed processes and by later removing them from the list in case it believes that the suspecting was a mistake.

Chandra and Toueg [10] characterized eight different classes of failure detectors in respect to its *completeness* and *accuracy*. Completeness requires that a failure detector eventually must suspect every process that crashed. Accuracy restricts the flexibility that the failure detector has to make mistakes (e.g. when a correct process suspects another correct process).

In respect to Completeness, there are two types:

- *Strong:* where every crashed processes is, eventually, permanently suspected by every correct processes.
- *Weak:* where every crashed processes is, eventually, permanently suspected by some correct processes.

Supplementary to Completeness there is Accuracy and this can be of four types:

- *Strong:* where correct processes are never suspected by any correct processes.
- *Weak:* where some correct processes are never suspected by any correct processes.
- *Eventual Strong:* where there is a time after which correct processes are not suspected by any correct processes.
- *Eventual Weak:* where there is a time after which some correct processes are not suspected by any correct processes.

In summary, with Chandra and Toueg [10] classification based on accuracy and completeness, failure detectors can be one of the categories shown on Table 2.2.

Types of Failure Detectors	**Completeness**	**Accuracy**
Perfect (P)	Strong	Strong
Eventually Perfect ($\diamond P$)	Strong	Eventually Strong
Strong (S)	Strong	Weak
Eventually Strong ($\diamond S$)	Strong	Eventually Weak
Weak (W)	Weak	Weak
Eventually Weak ($\diamond W$)	Weak	Eventually Weak
Quasi-Perfect (ϑ)	Weak	Strong
Eventually Quasi-Perfect ($\diamond\vartheta$)	Weak	Eventually Strong

Table 2.2: Types of Failure Detectors according to its level of completeness and accuracy

Chandra and Toueg [10] also defined how failure detectors can be equivalent with each other by exploring the concept of *reducibility*, where a failure detector D$'$ is *reducible* to a failure detector D if there is a distributed algorithm that can transform D into D$'$ which, in that case, D$'$ is said to be weaker than D. Thus, with reduction, anything that can

be done using the failure detector D′ can also be accomplished by using the D failure detector.

Dwork, Lynch, and Stockmeyer [11] introduced the concept of *partially synchronous* failure detector model, relaxing the asynchronous assumptions of the failure detector model and allowing a common notion of time between processes through the use of time-outs where one could assume that failed processes would trigger the algorithm about its failure given enough time.

2.4.1 Propagation of Failure Information

Acknowledging the implementation difficulties of a *partially synchronous* model, Felber, Défago, Guerraoui, *et al.* [12] identified how information about failure of a component is propagated through the system, i.e *flow policy*, and defined protocols such as *push model*, *pull model* and *push-pull model*.

In the push model protocol, *monitorable objects*, i.e. parameters than can be monitored, are active and periodically send *heartbeat* messages to inform that they are still alive. If a monitor, i.e. a failure detector, fails to receive any *heartbeat* from a *monitorable* object within a specific time bound, then it starts to suspect that the object has failed. After receiving a message from an observable, the monitor sets a timer that should trigger in case the next messages are not received. In this model only one-way messages are sent in the system and *multicast* can be used in case there are multiple monitors [12].

In the pull model protocol, monitor objects send *liveness* requests to monitored objects. When a *liveness* request is received, monitorable objects should reply informing the monitor that they are still alive [12].

In the push-pull model protocol, or *dual model*, the protocol has two phases. In the first phase, the monitor assumes that all of the monitored objects are using the push model, so it expects to receive *heartbeat* messages. On the second phase, the monitor switches the model for processes that didn't send *heartbeat* messages and assumes that such processes are using the pull model so they must be expecting for *liveness request* in

order to respond to the monitor. If no responses are sent to the monitor then the process is *suspected* [12].

2.5 Tools

Process communication through networks relies on datagram and streaming Application Programming Interfaces (APIs) and provides the base for building different communication protocols and higher-level communication systems. Following is an overview of the fundamental network communication methods that are present, whether indirectly, when they area abstracted by some other high level application, (such as the proxy or database layer), or directly, when there is as need to handle low-level details (such as timeouts, Internet Protocol (IP) addresses and port numbers).

2.5.1 TCP

The Transmission Control Protocol (TCP) is a communication protocol used for stream sockets. TCP provides a sophisticated transport service by supplying a reliable, connection-oriented, bidirectional, byte-stream communication channel and stream-based programming abstraction between two endpoints. A TCP endpoint is represented by the information maintained by the operating system for one end of a TCP connection as well as the send and receive buffers and the state information used to synchronize the operation between the two connected endpoints [13].

As TCP is connection-oriented, before any data transfer begins, the sending and receiving process need to establish a bidirectional reliable communication channel. Through this channel the processes can read and write to each other any time during the connection period, and intermediate nodes are aware of the TCP connections even though the IP packets carrying the data could follow different routes to its destination [1].

To guarantee that bytes sent by the sender process arrive without errors, and are assembled in the original order, TCP has error-detection and sequencing features. In order to handle errors, the data is broken into segments with a checksum so that the

receiver process can check if there is any error at some specific segment. When checking for segment errors the receiver endpoint must *acknowledge* when data is fine or discard it otherwise. If the sender endpoint does not receive any *acknowledgement* for a particular segment, due to errors or a timeout, then it must send that same segment again until it receives the *acknowledgment*.

To ensure the order of segments over a TCP connection, each segment is assigned a logical sequence number indicating the position of that segment in the data stream for the connection. With the sequence number, the receiver is able to assemble the TCP segments in the correct order and pass them as a byte stream to the application layer. The sequence number is also used as an acknowledgment to the sender so that both endpoints know exactly which segments have failed and needs to be sent again. TCP also has flow and congestion controls to prevent that the number of drop segments or segment re-sends increases due to the heterogeneity of the underlying network [13].

In conclusion, TCP is a reliable way of transmitting data between two endpoints. Due to TCP built-in error checking and resend of failed segments, it is a good tool for applications such as file transfer and games.

2.5.2 UDP

User Datagram Protocol (UDP) is a communication protocol used for datagram sockets. UDP is neither connection-oriented nor offers any kind of reliability. Thus the message could be corrupted or may not ever arrive at the destination. With UDP, if reliability is needed, it must be implemented at the application level [13].

To send or receive UDP messages a process must create a datagram socket bound to an IP address and a local port. When working as a server the process must *bind* its socket to a local port (i.e., select a particular port) where it could listen for some client messages sent to the server. A Client binds its socket to any free local port (i.e. *ephemeral port*) and every exchanged messages contains the IP address and port number, of both

the sender and the receiver, so that the transport-layer is aware of the communication endpoints and the client can reply to the server whenever needed [1].

Despite UDP intrinsic unreliability, there are some characteristics that makes it a valid option depending on the application domain:

- As UDP does not need to neither establish nor close a connection, it can be faster than TCP. When some error occurs UDP will not try to resend the invalid messages as in TCP [13].

- With UDP sockets is possible to use *broadcast* and *multicast* addressing methods. Broadcast permits a sender to transmit a datagram, to the same destination port, on every host of a connected network. Multicast allows a sender to transmit a datagram, to the same destination port, on a specified group (i.e. set of hosts) [13].

- Applications that can accept the loss or corruption of some messages such as audio and video streaming can be implemented with UDP.

2.5.3 Message Queues

Message Queue is an Inter-Process Communication (IPC) that provides a *point-to-point* service using the concept of a queue in which processes can consume messages, (e.g. allowing a processes to exchange data in the form of messages), and with that achieving space and time decoupling. A *producer* process can *send* message to a determined queue and some *consumer* process can *read* messages from this queue. Queues normally work on an First-In-First-Out (FIFO) order, but message queues implementations can support other policies such as priority, where higher-priority messages are delivered first [1]. With regard to client processes, the reading of a message from a queue can follow different styles:

- *Blocking Receive:* block until a message is available.

- *Non-Blocking Receive:* also referred as polling operation, constantly checks the status of the queue returning a message if available or some kind of unavailable indication otherwise.

- *Notify Operation:* an event is issued whenever a message is available for consumption on a queue.

Messages are persistent until its consumption, thus asserting that messages will, eventually, be delivered, although it is not possible to make assumptions about the precise time. Message Queue can also provide functionality such as: (i) *transactions:* where a message could be part of a set of steps that needs to be fully completed in an atomic fashion; (ii) *transformations:* where messages could be transformed to some other format and with that deal with heterogeneity of data representations; (iii) *security:* where implementations could offer support for authentication and confidentiality through Secure Sockets Layer (SSL) [1].

2.5.4 Group Communication

Group communication is a type of an indirect communication paradigm. Indirect communication represents a communication between a sender entity and a set of receivers through an intermediary element that avoids direct coupling by providing space and/or time uncoupling. With *space uncoupling*, as sender and receivers do not have each other identity, the system is flexible enough to support changes, update replications and migration of data. With *time uncoupling* neither sender nor receiver need to share the same life cycle, giving the system the ability to deal with entities that can be active or inactive. Time coupling generally means that messages should be persisted until the receiver (s) are ready to receive the messages [1].

In a Group Communication service, messages are sent to a named *group* and later are delivered to all members of the specified *group*. A *group* is an abstraction, generally implemented over IP multicast, where the sender does not need to know any receiver identity. Different implementations may have different group management policies, group

membership services and failure detection according to the system needs [1]. The group membership service has to deal with four main tasks:

- *Group changes:* provides an interface to create or destroy a group and also do add or remove processes from groups.
- *Failure Detection:* monitoring of group members when they crash but also when there is some communication failure; can mark a process as *Suspected* or *Unsuspected* whenever there is a communication failure.
- *Notification:* notify group members whenever there is some some change in the current state of the group (e.g. the group view has changed).
- *Address Expansion:* given a message and a group identifier, coordinates the multicast delivery of the message to all of the group members.

As groups work over multicast communication, only one send operation is needed by the application in order for the message to be sent to all the group members. This provides better usage of bandwidth and relieve system designers from the burden to have to deal with it [1].

2.5.5 Middleware

A Middleware is a software layer that provides a programming abstraction and masking of the heterogeneity characteristics of underlying layers such as networks, hardware, operating systems and programming languages, by providing a uniform programming model, such as Remote Procedure Call (RPC), Remote Method Invocation (RMI) among others, that can be used by both servers and distributed applications designers in a more homogeneous environment where different systems can interact with each other even if their underlying hardware and software characteristics are different [1].

Chapter 3

High-Availability Architecture and Benchmarks

This chapter introduces a high-availability architecture for database access that optimizes performance and replication between databases. Such architecture was conceived taking into account the possibility to replicate it in a real environment with available resources. Section 3.1 describes the architecture's individual elements. In Section 3.2, the different types of component replication are discussed. Section 3.3 discusses the benchmark, the underlying infrastructure and what software tools were used to collect performance metrics from the system. Section 3.4 presents the benchmark results and threats to the validity of the approach.

3.1 Architecture Overview

A high availability system architecture was conceived to gain insight into the implications of such a system in a real scenario.

In the proposed architecture, shown in Figure 3.1, some application needs to access a persistence layer and connects to it through a proxy layer, eliminating the need for it to know about the internal topology of the persistence layer and/or its writable or readable replicas.

The data storage layer, in the context of this work, is made of four individual instances of the MariaDB Relational Database Management System (RDBMS). The proxy layer, with its two Apache Sharding Sphere instances, acts as a load balancer and failover layer where, in case of a primary proxy failure, the secondary proxy could assume its role until the primary becomes available again.

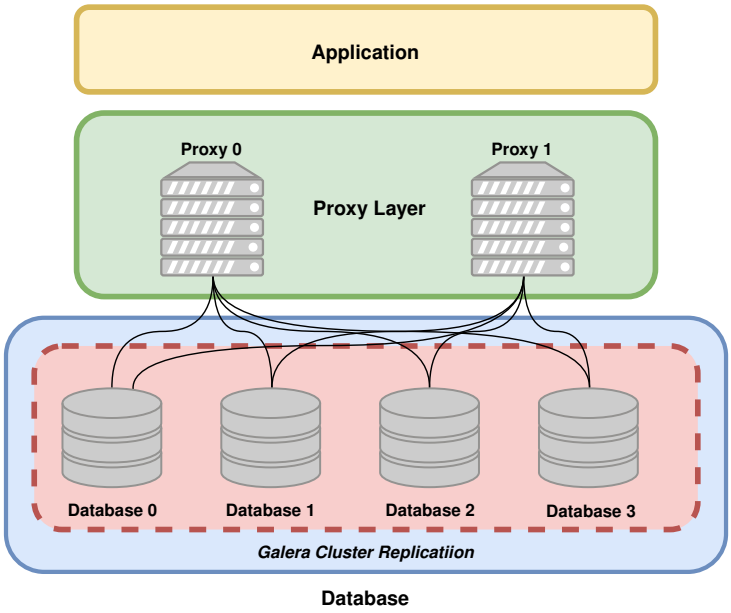

Figure 3.1: High-Available Architecture

3.1.1 Apache ShardingSphere

Apache ShardingSphere is an open-source project that consist of a set of distributed database middleware solutions namely: *Sharding-JDBC*, *Sharding-Proxy* and *Sharding-Sidecar*. The goal of the ShardingSphere project is to provide functions for data sharding, distributed transactions an database orchestration. *Sharding-JDBC* is an enhanced Java Database Connectivity (JDBC) that is focused on high performance while maintaining compatibility with all JDBC drivers frameworks. *Sharding-Proxy* is a transparent database proxy, that provides a database server that encapsulates database binary protocols in order to support heterogeneous languages. It is possible to configure sharding

without the need to apply vendor specific configurations to server databases. *Sharding-Sidecar*, while still under development, is a cloud native database agent responsible for all access to database in the form of a DaemonSet (Kubernets) for container platforms. It provides a decentralized mesh layer that interacts with databases, i.e. a Database Mesh or database grid [14].

Other proxy softwares were considered for this projects such as ProxySQL [15], developed in the C++ programming language; but, as the failure detector is developed in Java programming language, the adoption of ProxySQL would require more complexity on its integration with the failure detector as they are from distinct ecosystems. Apache ShardingSphere was then chosen to reduce technical complexity by maintaining a single development ecosystem as both the failure detector and the ShardingSphere proxy are developed in Java programming language.

3.1.2 MariaDB

MariaDB is an open source RDBMS written in C and C++. It began as a fork of MySQL after it was acquired by Oracle. As other RDBMS, MariaDB permits the creation and management of relational databases by using Structured Query Language (SQL) queries directly or by integration with some external applications. MariaDB has built-in support for disaster recovering and high availability through fail-over and replication features. On the security aspect MariaDB offers support for data encryption and even data obfuscation for anonymization [16].

3.2 Replication Methods

By analyzing the architecture shown in Figure 3.1, when it comes to the application communicating to the persistence layer, it is possible to note that all components have replicas and there is not a single point of failure. Albeit this seems to suffice for a high-available system there is also the need for a well defined course of action in case of a

component failure. Such failure detection and response behavior is discussed in more details for each layer below.

3.2.1 Proxy replication

The proxy layer is composed of Apache Sharding Sphere instances that need to be aware of the persistence layer and each database instance. One individual proxy instance, originally, is not aware of other proxy instances and failure detection and response measures needs to be done through multiple Domain Name System (DNS) records or virtual IP.

Without the failure detector, proxy instances are unable to detect database failures causing the system to crash in subsequent accesses to a failed database. To mitigate such problem, the failure detector module was later added to the proxy and databases in a way that in the event of a database failure, the proxies, or even the other databases, could dynamically activate or deactivate the failed database instances until it becomes functioning again. This approach, albeit not perfect, contributes to the overall reliability and availability of the system by reducing the impact and propagation of a failed component as long as there are the minimum resources needed to guarantee the system functionality.

3.2.2 Database replication

At the persistence layer there are three different methods to achieve redundancy, namely the **asynchronous method**, the **semi-synchronous method** and the **synchronous method**.

On the asynchronous method only one database, the master, receive write statements while it commits its changes to a binary log that later is read by the other databases working as replicas. Such method, shown in Figure 3.2 (a), is good for read operations, as adding new replicas does not add load to the whole system, but it is prone to information loss in case the master fails and the replicas failed to complete the syncing process. With the asynchronous method there is not failover and in case of a failure of the master another master must be chosen manually from the set of replicas.

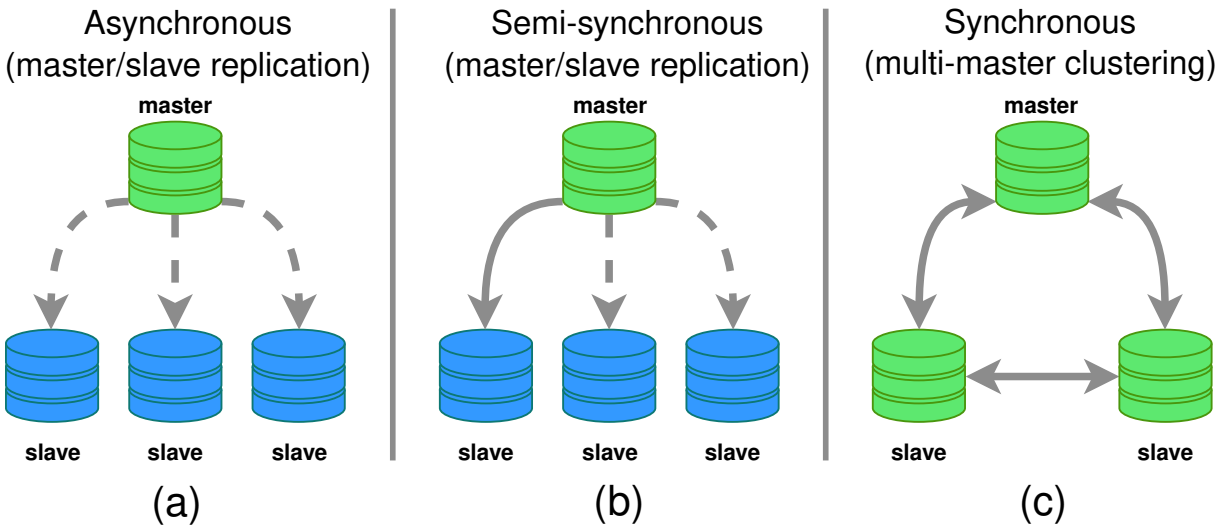

Figure 3.2: Replication types (Adapted from MariaDB [17])

On the semi-synchronous method, a transaction is considered committed only after it is committed in, at least, one of the replicas. Such method, shown in Figure 3.2 (b), reduces the probability of data loss. In case of a failure, there still lacks an automatic failover mechanism, so it is necessary to manually intervene, with the difference that only the replicas synced with the master should be able to become the new master. This adds more complexity to the maintainability of the system as there are 3 states to handle: master, replica and semi-master.

Finally, on the synchronous method, all databases work as a master and there is no need to transfer logs. In this project, such was achieved by, Galera Cluster [17] was used to manage the synchronous replication throughout MariaDB nodes. The synchronization happens by propagating changes, such as locks and data that needs to be replicated, as they come, to other nodes through a dedicated communication channel. Galera Cluster, shown in Figure 3.2 (c), uses a quorum-based failure detection system, and in order to work properly and mitigate data inconsistency, it needs at least 3 nodes. That way, Galera Cluster failure detection mechanism can apply the quorum-based state voting and re-sync with other nodes as needed. This method is good for write and reading scaling as is possible to use all of the nodes without the need to manually choose between node roles.

Galera Cluster

Galera Cluster [17], developed by Codeship, is a clustering solution for MySQL and MariaDB databases. Galera Cluster offers support to synchronous multi-master database replication based on InnoDB storage Engine. Multi-master clustering allows writes to happen to any server that is part of the cluster without the need to deal with distributed locking and shared resources management.

In practice a Galera Cluster consists in several database instances, (MySQL or MariaDB) with the Galera Replication Plugin installed. This plugin implements the Write-Set Replication (WSREP) API, which provides a certification-based replication as a transaction for replication, i.e. write-set, with rows to replicate and locks that were held by database, sent between the nodes in the cluster. Each node certificates that the write-set was replicated to other nodes and, at this is stage, the transaction is considered committed. In a scalability perspective the Galera Cluster also helps to boost performance by allowing writes to every node in the cluster at the same time, as all nodes holds the same data [17][18].

3.3 Benchmarks

Benchmarks makes possible to validate and collect metrics of a system performance. A benchmark tool provides an environment that is stable, controlled and repeatable while being flexible enough to handle different sets of configurations. This can help to analyze systems behavior, and potential bottlenecks, when subjected to different kinds of workload.

To better understand the limitations and performance of the infrastructure the architecture introduced on Figure 3.1 has been benchmarked with the TPC Benchmark C (TPCC) [19].

3.3.1 TPC Benchmark C

TPCC generates and evaluates an Online Transaction Processing (OLTP) workload. It is a mixture of read-only and update intensive transactions that simulate the activities found in complex OLTP application environments such as *e-commerce*. It does so by exercising all of system components associated with such environments, which are characterized by:

- The simultaneous execution of multiple transaction types that explore different complexities.
- On-line and deferred transaction execution modes.
- Multiple on-line terminal sessions.
- Moderate system and application execution time.
- Significant disk input/output.
- Transaction integrity (ACID properties).
- Non-uniform distribution of data access through primary and secondary key.
- Databases consisting of many tables with a wide variety of sizes, attributes, and relationships.
- Contention on data access and update.

The performance metric reported by TPCC is a "business throughput" measuring the number of orders processed per minute. Multiple transactions are used to simulate the business activity of processing an order, and each transaction is subject to a response time constraint. The performance metric for this benchmark is expressed in Transactions per minute-C (tpmC).

3.3.2 OLTPBenchmark

OLTPBenchmark is a multi-threaded load generator framework. The framework is designed to be able to produce a variable mixture load against any JDBC-enabled relational database. The framework also provides data collection features, e.g., per-transaction-type latency and throughput logs [20]. The OLTPBenchmark tool was chosen to run the benchmarks due to its configuration flexibility, implementation of the TPCC Benchmark and also for being database agnostic.

OLTPBenchmark has built-in support for the following benchmarks: TPCC, Wikipedia, Synthetic Resource Stresser, Twitter, Epinions.com, TATP, AuctionMark, SEATS, YCSB, JPAB (Hibernate), CH-benchmark, Voter, SIBench (Snapshot Isolation), SmallBank, LinkBench.

Each benchmark can be parameterized through an Extensible Markup Language (XML) file. On Appendix 1, the configuration used for the tests in this project is shown, with the first 7 tags being for driver and connection specification as well as the level of transaction isolation that database sessions should be open and following remaining tags being TPCC specific. The `<scalefactor>` tag controls the size of the database. The `<terminals>` tag controls the workload, or the number of clients, that would be generating traffic in the system. The `<works>` tag is used to describe different types of tests that OLTPBenchmark would perform by configuring the transaction rate (1000 in this case), the warm up time before any metrics is done (100 in this case), and the time that the test should be measured (200 in this case).

3.3.3 Test Scenarios

The test infrastructure is composed of 7 Virtual Machines (VMs) with 3.85 GB of RAM, 63 GB of SSD and 4 cores AMD EPYC 2.4 GHz processors each running Debian GNU/Linux 9 (stretch) as the operational system. Each machine has one specific service or has one specific role and they were distributed as follow:

- 4 VMs to compose the persistence layer by running instances of MariaDB Version 15.1 Distrib 10.1.37-MariaDB.
- 2 VMs to compose the proxy layer and dedicated to run Sharding Sphere proxy instances.
- 1 VM is dedicated to generate the workload of the system through OLTPBenchmark.

All VMs were part of the same dedicated logical network (a specific VLAN) and were interconnected by a single switch, as shown in Figure 3.3. Ping time of 0.529ms, 0.793ms, 1.784ms and 0.220ms for minimal, average, maximum and deviation, respectively, were measured between each VM.

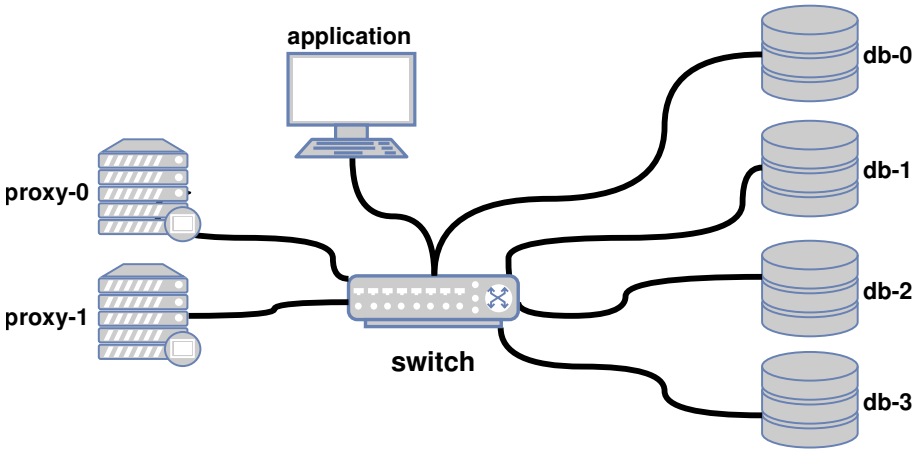

Figure 3.3: Virtual Machines infrastructure

The goal of the benchmarks on the high available architecture, introduced on Figure 3.1, is to get an overview of the its performance behaviour while exploring different database layouts, by varying how many instances could be written or read from. Such layouts are discussed in more details below.

Application connects directly to a database

The purpose of this layout, where an application bypasses the proxy by connecting directly to a database, as shown on Figure 3.4, was to later be able to compare a direct connection with a connection behind the proxy and spot any performance drawbacks.

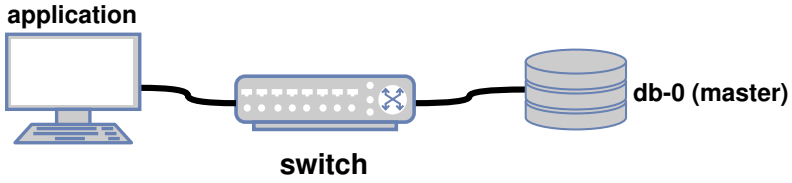

Figure 3.4: Test scenario using JDBC directly to a database

Application connects to a proxy with single database

The purpose of this second test scenario, shown in Figure 3.5, was to verify the performance drawbacks that could exist by using a proxy as a middle layer between the application and the database layer with a single database to write and read from.

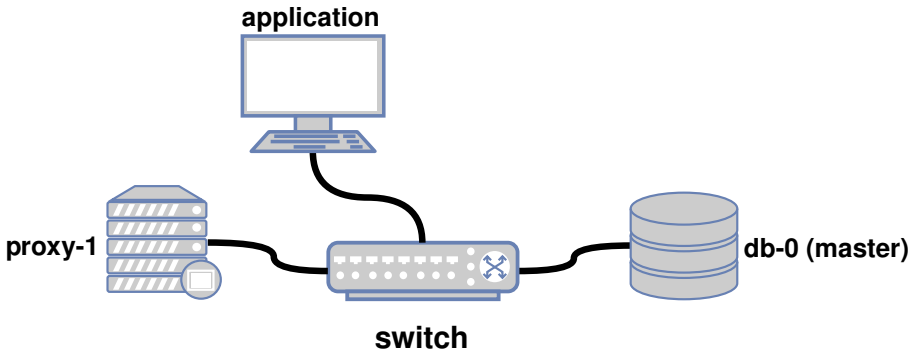

Figure 3.5: Test scenario using a proxy with one database

Application and Proxy with one master and one read replica

This third test scenario, as shown in Figure 3.6, was proposed to test how the system could perform when writing to a single database instance (master) while being able to read from two instances (one master and one replica).

Application with Proxy with two masters and two read replicas

This test scenario, shown in Figure 3.7, was proposed to test how the system could perform when writing to two database instances (multi-masters) while being able to read from other two database instances (replicas).

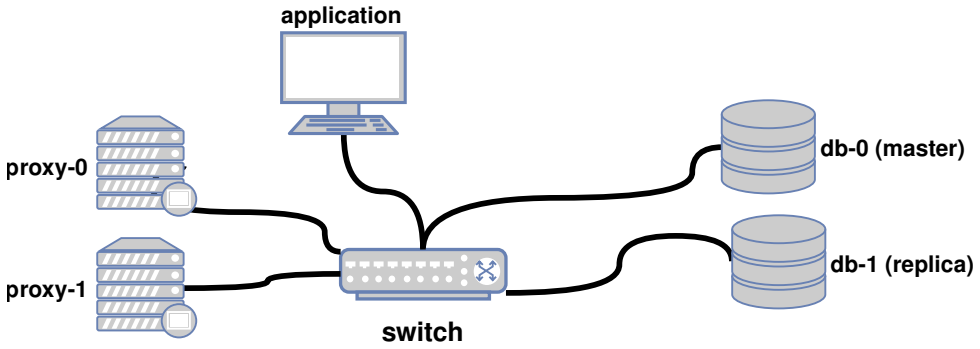

Figure 3.6: Test scenario using a proxy with one master and one replica

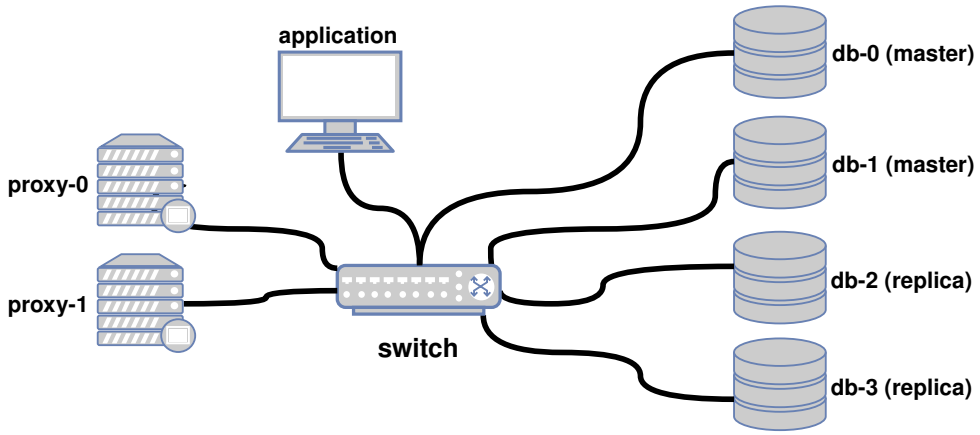

Figure 3.7: Test scenario using a proxy with 2 masters and 2 replicas

3.3.4 Database configurations

Besides changing the layout of the architecture to test different behavior of the system according to the use of database resources, by benchmarking it was identified that with default configurations, MariaDB databases were not being stressed to its full capacity on each machine. Such behavior negatively affected the initial tests as instances ended up limiting and decreasing performance due to poor database configuration. To increase performance some adjustments were made on parameters such as the number of connections; transaction and statements timeout; concurrency handling; buffer and cache size of the data storage engine. The final configuration file can be verified at Appendix 2.

3.3.5 Running the Benchmarks

To analyze the architecture with different workloads the benchmarks were run by varying the number of concurrent clients starting with 1, 2 and 4 threads and from that up to 128 threads, with 4 thread steps.

3.4 Results

After running the benchmarks with each of the different scenarios and varying the workload, the tpmC were collected and plotted on the graph shown on Figure 3.8. The horizontal axis, named Terminals, represents the number of concurrent clients and, the vertical axis, named Throughput, represents the number of transactions. Each line of the graph represents the throughput achieved by a different configuration layout for a different number of active concurrently clients (terminals).

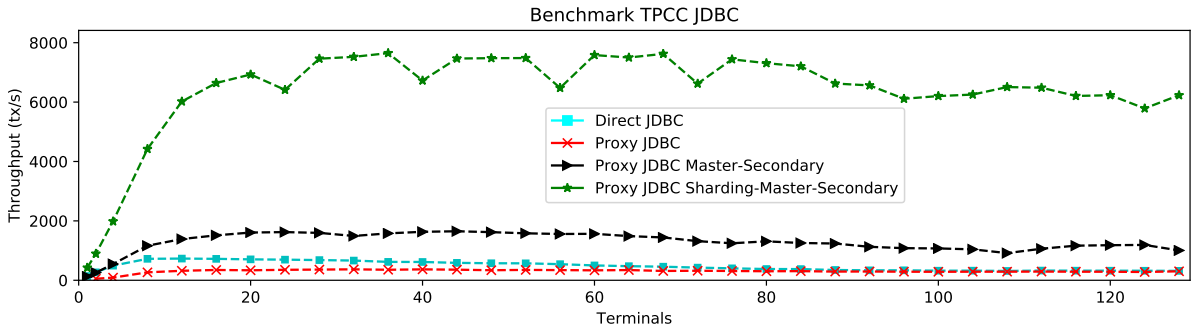

Figure 3.8: Benchmark results

By observing the Figure 3.8 is possible to see that for the Direct JDBC line, which represents the configuration where the application connects directly to a single database thus ignoring the proxy, the throughput reached its peak around 430 tx/s, and by comparing with the Proxy JDBC line is possible to see that, in fact, the proxy adds a considerable overhead (twice approximately) to the structure and had its throughput peak around 200 tx/s.

Comparing with other lines, Proxy JDBC Master-Secondary and Proxy JDBC Sharding-Master-Secondary, it is possible to see how the proxy could help with performance by

increasing the number of write and read instances. With Proxy JDBC Master-Secondary writes happens exclusively at one database instance and reads happens exclusive on the other database instance reaching its throughput around 1600 tx/s. Though, on Proxy JDBC Sharding-Master-Secondary, when reading from 2 instances and writing to other two instances the throughput reaches its peak at 7520 tx/s.

3.5 Threats to Validity

Although databases were configured in a production fashion and the environment was controlled to its best, there are some important considerations about the results:

- All VM were sharing the same storage through a unique ZFS+NAS partition that was later subdivided for each individual virtual machine.
- All VMs shared a dedicated a 128 GB RAM for cache. Such amount would be more than enough to hold all the data from tables used on the benchmark. This could had a significant impact in the final results as the need for round trips to the disk would be reduced or even eliminated.
- Each node has a network transmission capacity of 1 Gbit/s and were in the same environment. On a considerable more physically distributed environment database synchronization and access could have a more unstable response time.

Such observations are relevant as they could have had a considerable impact on the performance gains seen by increasing the number simultaneous reading and writing instances.

Chapter 4

Implementation of an Eventually Perfect Failure Detector

As discussed on Chapter 2, the asynchronous characteristics of distributed systems poses a challenge to software design and implementation due to the lack of global timing between components which makes impossible to determine if a process has crashed (failed) or if it is taking a considerable amount of time to process some input. With this problems in mind, Chandra and Toueg [10], presented the concept of unreliable failure detectors that permit some degrees of mistake in order to eventually reach synchrony.

This chapter presents the design and implementation details of a Eventually Perfect Failure Detector using the Java Programming Language as well as the difficulties found during the implementation.

4.1 Build and Implementation Tools

This section presents an overview of the tools used to control the artifacts generated during the implementation of this project.

4.1.1 Java Programming Language

Java is an object-oriented, general-purpose programming language [21]. It was designed based on the concept of Write Once, Run Anywhere (WORA) by adding a dependency of the Java Virtual Machine (JVM). Java code is compiled to a *Bytecode* format that can run on any JVM regardless of the underlying computer architecture and/or operating system.

The failure detector was implemented in Java due to its complete network API and vast libraries that provides different distributed systems programming models through an object-oriented programming abstraction.

4.1.2 Gradle

Gradle is an Open Source build automation tool focused on flexibility, by offering support to expand itself, and performance by avoiding unnecessary work, running only tasks when their inputs or outputs have changed. As Gradle runs on JVM it is possible to take advantage of the whole Java API and run Gradle on a variety of platforms due to Java extensive compatibility.

The Gradle building model is based on a set of tasks, or units of work, connected by a Directed Acyclic Graph (DAG) that describes all of the dependencies and the order of execution. Tasks can be classified as: *Actions:* for works such as copying or compiling files; *Inputs:* values, files and directories that actions use or operate on; *Outputs:* files and directories that the actions modify or generate [22].

4.1.3 JFrog Artifactory

Artifactory is a JFrog product created to work as a binary repository manager [23]. A binary repository optimizes the software building process by storing the artifacts generated, often on a binary format, by the building process. Artifactory centralizes this management process and provides resources to better search, by adding meta-data to binaries

and guarantee reliability of such artifacts by proving replication and security measures such as authentication and encryption.

4.1.4 Apache ActiveMQ

Apache ActiveMQ is an open source software to build a multi-protocol, embedded, very high performance, clustered, asynchronous messaging system. Apache ActiveMQ is an example of Message Oriented Middleware (MoM), i.e. messaging system [24]. Messaging system helps to reduce the heterogeneity between different systems on the network, and to provide reliable, high available and an asynchronous messaging system. With its configuration flexibility, Apache ActiveMQ has support for transport protocols namely: OpenWire, Stomp, MQTT, AMQP, REST and WebSockets.

4.2 Design

Figure 4.1 represents a high level overview of the failure detector architecture and exposes its internal logical units as individual blocks. At the base level there is the **Failure detector** block which represents an unit of a failure detector process. A failure detector process should be able to communicate to other failure detector processes in order to build the knowledge about the system health as a whole and keep track of their individual process health status, i.e. to know if a process has crashed or it is just taking some time doing some computations.

Above the failure detector block there are the **Channel** and **Probes** blocks. The Channel should provide a common API and serve as a conduit through which failure detector processes can exchange messages, e.g. heartbeat messages, by using any kind of communication method that can comply with the channel API. As an example there is the **UDP** block in which processes communicate through UDP sockets, and there is the **ActiveMQ** block in which processes exchange messages through message queues (ActiveMQ in this case).

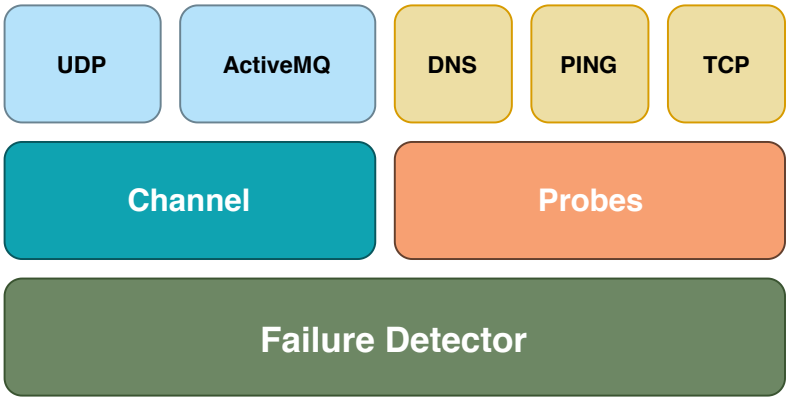

Figure 4.1: Failure detector block diagram

The Probes block should provide a common API for monitoring performance and checking the health status of other system components by sending probe tests through an end-to-end communication. A Probe test could be an e-mail message to test a mail service; a web request to test a web server; a database query; *ping* or *traceroute* to test network availability for example. By using test probes, the system can diagnosis the root-cause of a performance degradation and take some necessary action.

4.3 Implementation details

The block diagram gave a high level overview of the individual logical units and its responsibilities identified on the design phase. In this section the implementation aspects are discussed in a more practical view, by presenting class diagrams and snippets of code, for each logical unit, whenever necessary.

4.3.1 Failure Detector

The failure detector logical unit represents all of the essential features and behaviors of a failure detector. By extracting those common features and behaviors, a class diagram, as shown in Figure 4.2, was created. This diagram represents how the failure detector module was implemented in this project.

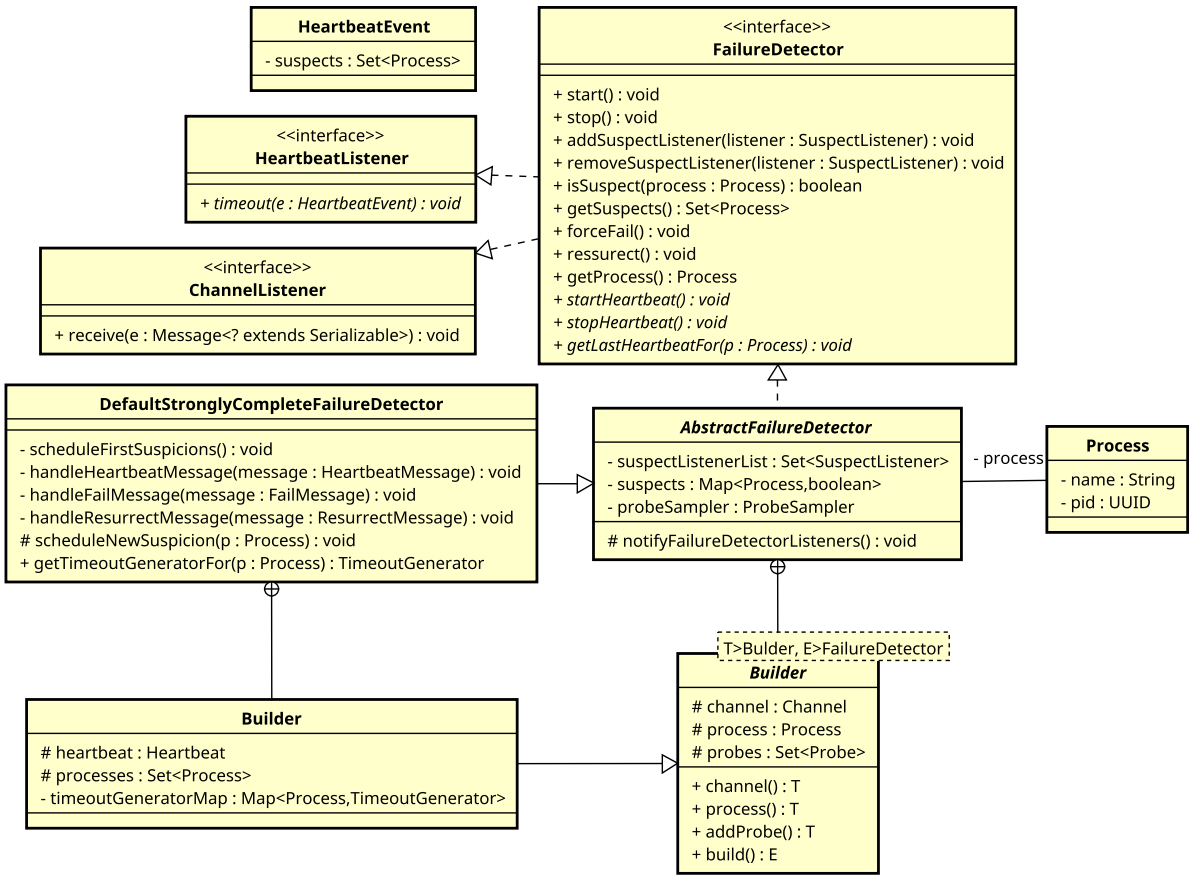

Figure 4.2: Failure detector class diagram

The fundamental functionalities of a failure detectors are represented by the **FailureDetector** interface. Its methods are:

- *start* and *stop*: methods to start and stop, respectively, the failure detector as well as its internal services.

- *startHeartbeat* and *stopHeartbeat*: methods responsible for starting and stoping, respectively, the heartbeat service of the failure detector.

- *addSuspectListener* and *removeSuspectListener*: methods responsible, respectively, for adding or removing an object from the list of processes to be notified in case there is an update of the suspects by the failure detector. Whenever this event is fired all

of the SuspectListener objects receive, from the failure detector, the updated set of the currently suspected processes.

- *isSuspect*: method to check if a given process is being suspected by the failure detector.
- *getLastHeartbeatFor*: get a **Date** object representing when was the last heartbeat of a given process.
- *getSuspects*: get the set of currently suspected processes.
- *getProcess*: get the process that represents the instance of the failure detector. Each failure detector has a unique process representation.
- *forceFail*: method to force the failure detector to enter a failure state by broadcasting a **FailMessage** to all peers listening on the channel. Each failure detector that receives such message should add the source process into the set of suspected process.
- *ressurect*: method for the failure detector to inform that it is working correctly by broadcasting a **RessurectMessage** to all peers listening on the channel. Each failure detector that receives such message should remove the source process from the set of suspected process.

The responsibility of maintaining the suspects list, and notify other failure detector processes about failures, is taken by the abstract class **AbstractFailureDetector**. This class also maintains a reference to a **ProbeSampler** object that handles the probes activity. The class **AbstractFailureDetector** also provides a way to better control the construction process of the object through the Builder pattern.

Heartbeat

Following the push model described by Felber, Défago, Guerraoui, *et al.* [12], this implementation of an Eventually Perfect Failure Detector ($\diamond P$) uses heartbeats to inform about the functioning state. At some specified amount of time, the failure detector should

broadcast heartbeat messages to other failure detector processes. In case of omission of those messages, other failure detectors will start to suspect that the process has failed. Figure 4.3 presents a class diagram of the classes involved on the failure detector heartbeat activity.

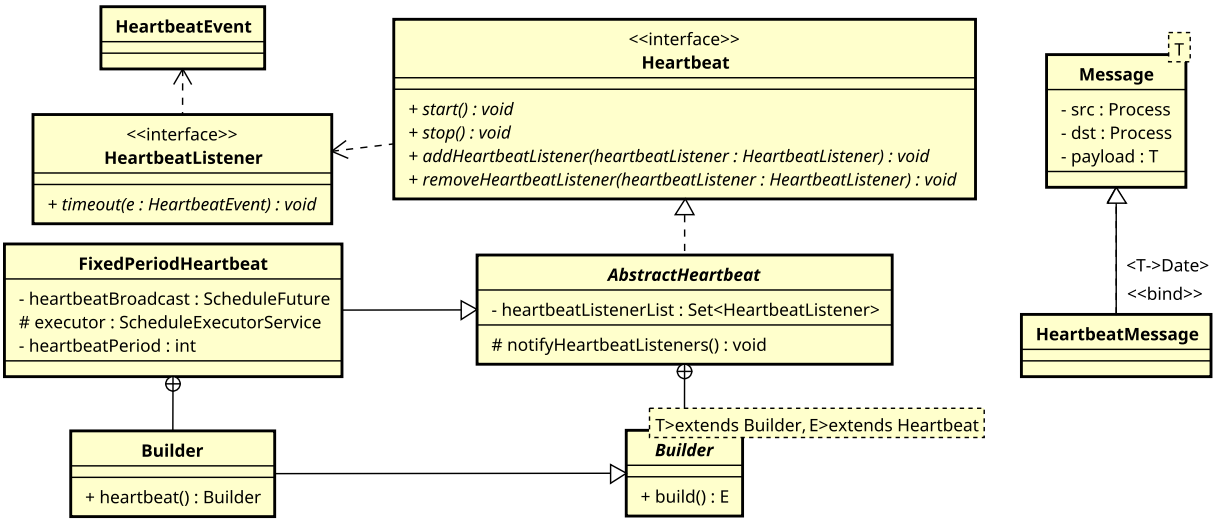

Figure 4.3: Heartbeat class diagram

As shown on Figure 4.3, by implementing the **Heartbeat** interface and notifying the **HearbeatListeners**, and by using the **builder** pattern, the abstract class **AbstractHeartbeat** makes it easy to implement different kinds of heartbeat behaviour.

The **FixedPeriodHeartbeat**, for example, implements the heartbeat behavior with a fixed period of timeout. A **ScheduledExecutorService**, from Java 8 and **java.utils** API, schedule the call of the **notifyHeartbeatListeners** method from **AbstractHeartbeat** with a fix period of time. Whenever a **HeartbeatListener** receives such notification it should fire its **timeout (e: HeartbeatEvent)** method, causing the failure detector to broadcast instances of the class **HeartbeatMessages** to all of its known failure detector processes, thus indicating that it is still alive.

Timeout

This implementation of the failure detector starts suspecting a process in a case a given period of time has passed with no heartbeat message received from the given process. To

better control how the failure detector would handle such scenario the timeout classes, as shown on Figure 4.4, were implemented.

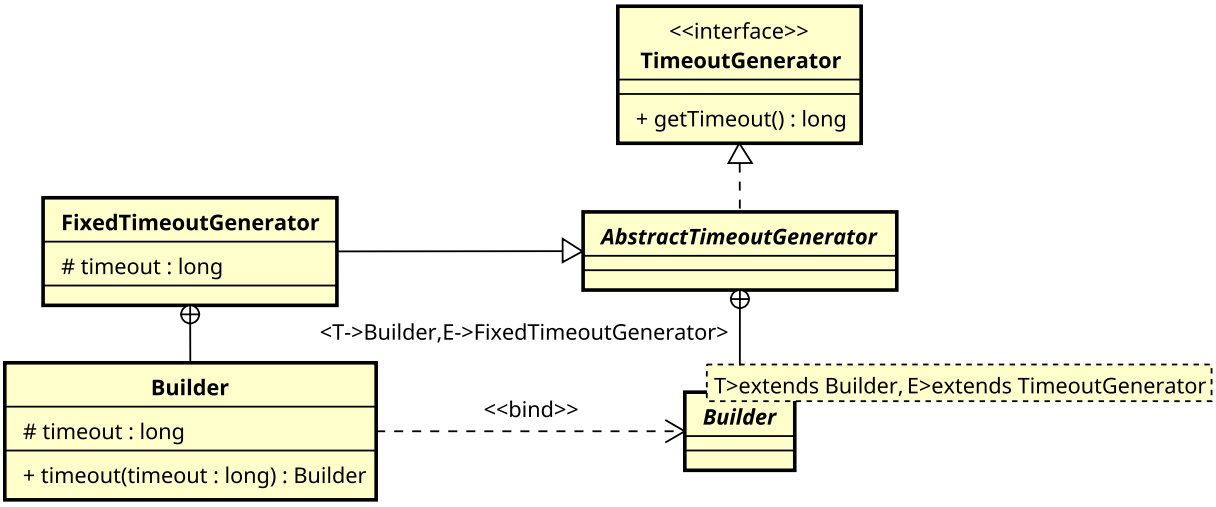

Figure 4.4: Timeout class diagram

By implementing the **TimeoutGenerator** interface and relying on the **Builder** pattern the **AbstractTimeoutGenerator** abstract class provides a flexible way to implement different timeout policies that could, for example, be based on the mean time of heartbeat messages, the longest timeout seen and so on. Failure detectors call the **getTimeout ()** of a class whenever it needs a timeout value to trigger a new suspicion.

The **FixedTimeoutGenerator** class, uses a fixed timeout value passed through the **timeout (int timeout)** method of the **Builder**. The final timeout, that will actually be used, is then calculated by $\text{timeout} = 2 * \text{timeout} + 1$.

4.3.2 Channel

The **FailureDetector** interface extends the **ChannelListener** interface in order to be able to receive messages from the channel through the *receive* method. The Figure 4.5 gives a better view of the responsibilities of the channel as well as the types of messages that could be exchanged on the channel.

The channel interface is responsible to provide a common way to start and stop a communication channel, as well as to provide the ability to interact with it by sending

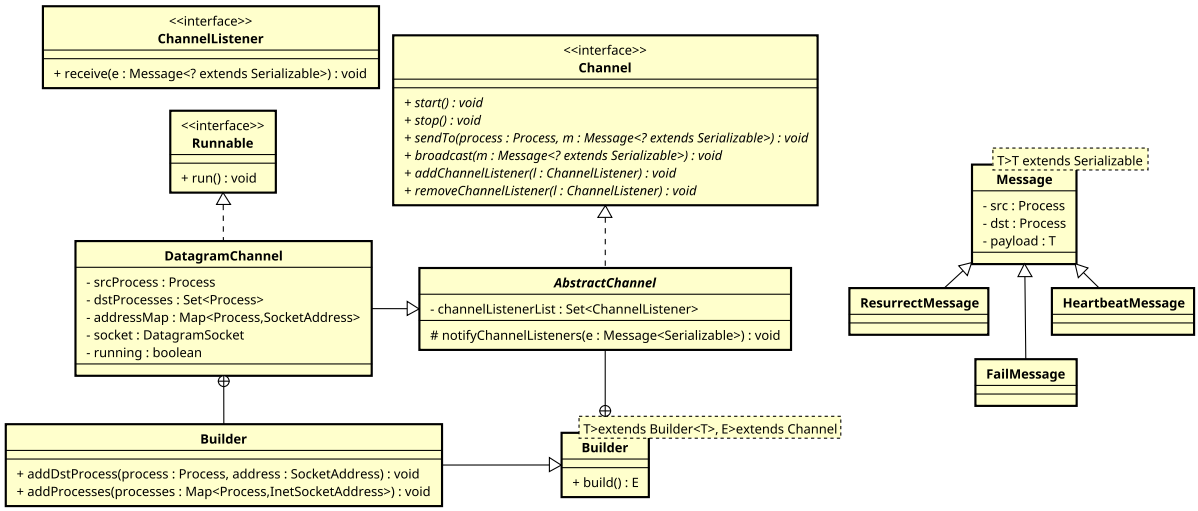

Figure 4.5: Channel class diagram

messages and registering other objects that implements the *ChannelListener* interface, to listen when new messages arrive on the channel. The abstract class **AbstractChannel** is responsible to maintain the set of channel listeners and notify whenever there are new messages. This class also provides a **Builder** inner class so that concrete implementations, such as the **DatagramChannel** have more control over the building process. The **DatagramChannel** inherits the **AbstractChannel** and provides a communication channel over UDP sockets. Other processes are added to the channel at the building phase and the the channel keeps a map data structure with each process mapped to its IP address and port number.

As shown in Figure 4.5, there are three types of messages, all of which inherits the **Message** class, namely **FailMessage**, **ResurrectMessage** and **HeartbeatMessage**. The **FailMessage** is used by a failure detector to force its failure, signalizing that it should be considered to be failed by other failure detectors. The **ResurrectMessage** is used by a failure detector to signal that its process should be considered alive by other failure detectors and removed from the set of suspected processes. The **HeartbeatMessage** is used by a failure detector to signal that its process is still alive. In case of omission then the failure detectors starts to suspect the missing heartbeat process.

4.3.3 Probes

The Probes block should provide a common API for checking, i.e. probing, the *liveness* of other system components. Such components could be a Domain Name System (DNS) resolution, test reachability of a host through ping commands, and TCP connection establishment capabilities, for example. The liveness is a metric to let the system know the health status of a system component by certifying that the given output is as expected to be.

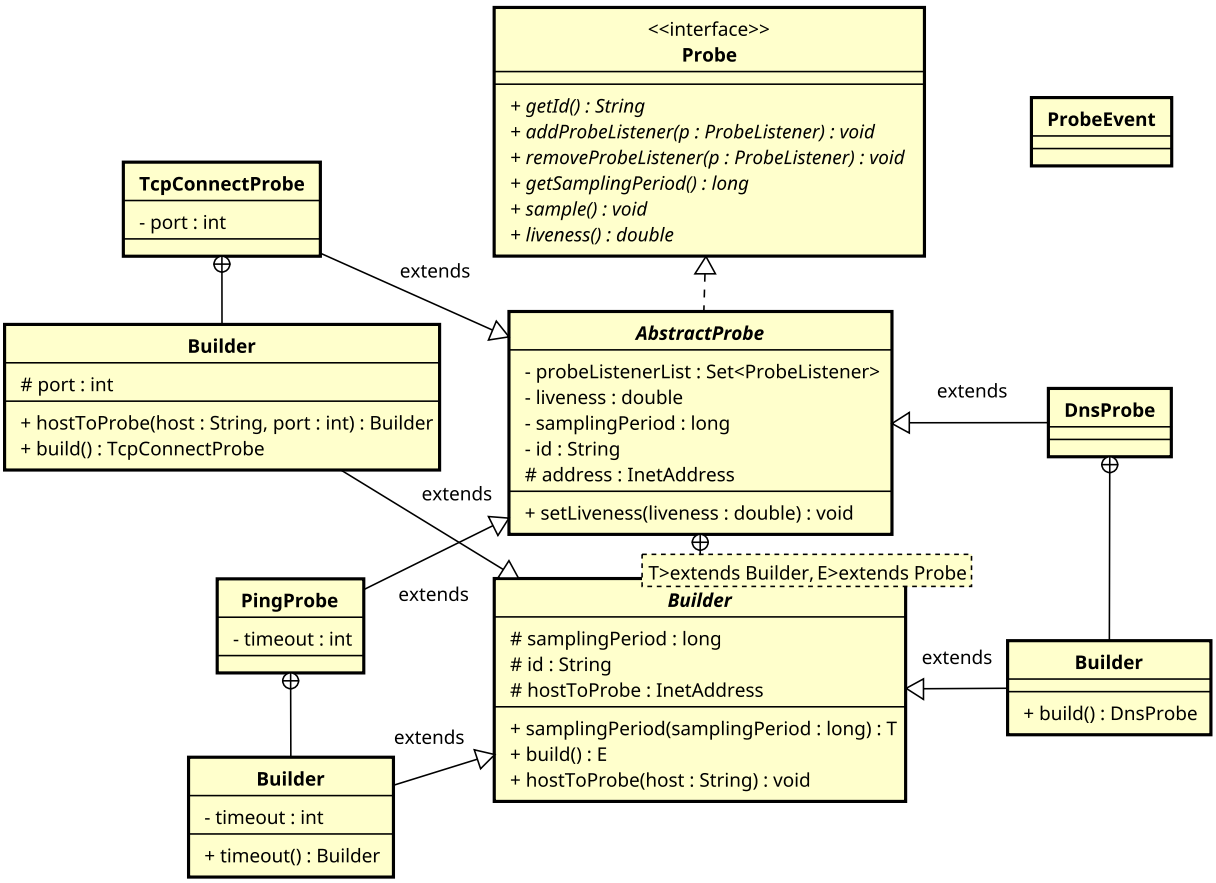

Figure 4.6: Probe class diagram

Figure 4.6 represents the class diagram of how the probe logic was implemented. The common attributes and methods that a Probe should have are described by the **AbstractProbe** class and the **Probe** interface.

A concrete implementation of an **AbstractProbe** abstract class, needs to implement the following methods:

- *getId* to get an identifier for the probe.
- *addProbeListener* and *removeProbeListener*: to register, or remove, an object to receive a notification whenever a **ProbeEvent** is generated due to liveness update.
- *getSamplingPeriod* gets the period in which the probe do its sampling activity.
- *sample* defines what would be the probe sampling behaviour. It's necessary that at the end of the sampling the probe update the liveness status of the component or aspect sampled. The values range from $\{0, 1\}$ where 0 represents a fail during the sampling and 1 represents a successful sample.
- *liveness* gets the currently liveness status from the probe. As said, is a value that ranges from $\{0, 1\}$.

4.3.4 Curator

The curator class, as shown in Figure 4.7, was created to control what data sources are available to be written or read by interacting with Apache Zookeeper and dynamically change proxy configurations. Its **dataSourcePaths** and **schemaName** attributes holds information about the proxy and its known data sources. Bellow is an overview of the methods from **Curator** class:

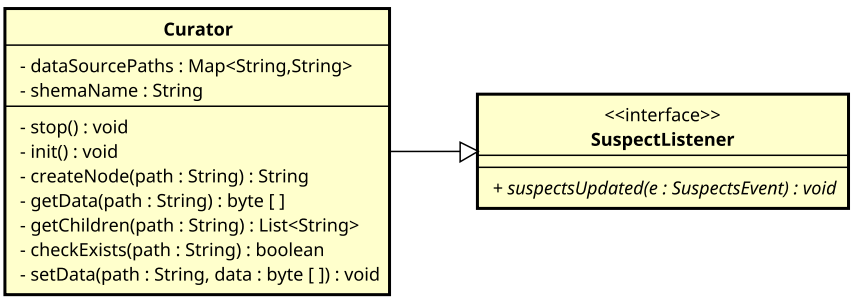

Figure 4.7: Curator class diagram

- *stop* and *start*: methods to start and stop the communication with Apache Zookeeper.
- *createNode*: creates a persistent node on Zookeeper on the given path.
- *getData*: get the data associated with the given path.
- *getChildren*: get all the children nodes of the parent node located at the given path.
- *checkExists*: checks if a given path exists on the Zookeeper.
- *setData*: associate data to a given path.

4.4 Architecture with Failure Detector

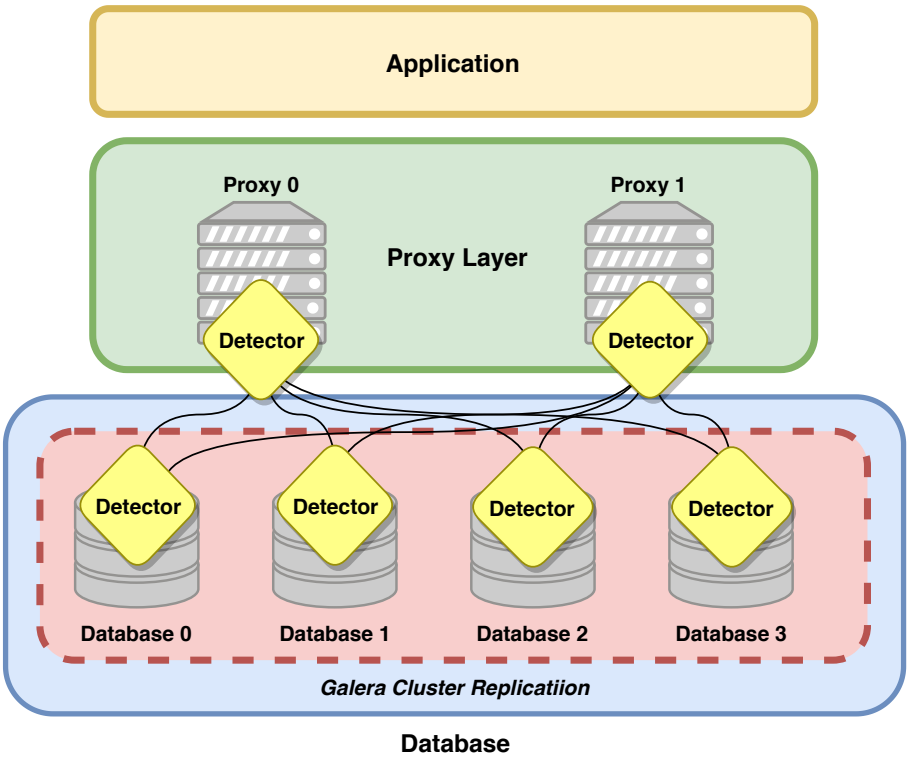

Figure 4.8: The architecture with Failure Detector

Figure 4.8 shows the final architecture but with the failure detector operating together on the database and proxy instances. Each failure detector instance is aware of

all instances and are constantly sending heartbeats and suspecting processes accordingly. Whenever some database unit is suspected, the failure detector deactivates its datasource until the process is being suspected.

With future implementation of SQL probes, it would be possible to remove failure detector instances from the database and decide if it has failed based on diagnosis of probe samples.

Chapter 5

Conclusion and Future work

This work introduced an implementation of an Eventually Perfect Failure Detector for a High Available system based on the fundamental theories and its integration with a proxy and database instances.

The initial effort was put in the conception of a High Available architecture for database access through a proxy. First benchmarks on the proposed architecture had shown that, by using a proxy, the performance could increase with the number of dedicated primaries (write) and secondaries (read) databases and paralelization of client transactions.

Implementation of the failure detector had a modular design, aiming to facilitate later addition of different kinds of communication channels, probes and timeout strategies for the heartbeat and suspicion phase.

Further testing had shown that the failure detector, when added to the architecture, increased the availability of the systems as failed nodes were dynamically removed or added in case of becoming functional again, thus masking, when possible, a database failure to the application accessing it.

In conclusion, by running the initial benchmarks and integrating the failure detector to it shows evidence that (i) performance could be considerably improved by replicating the number of database instances to read and write from, and (ii) using a failure detector helps with the availability in case of database failures.

For future work, besides refactoring and simplifying the failure detector source code, the replication logic could be also implemented by the proxy layer through a cache mechanism and more tests should be done with sharding databases in order to check if consistency and performance improvements happens in such complex cases as well.

Bibliography

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems*, 5 edition. Pearson, May 2011, ISBN: 978-0-13-214301-1.

[2] C. H. Lie, C. L. Hwang, and F. A. Tillman, "Availability of maintained systems: A state-of-the-art survey", *AIIE Transactions*, Jul. 2007. DOI: 10.1080/05695557708975153. [Online]. Available: https://www.tandfonline.com/doi/pdf/10.1080/05695557708975153?needAccess=true.

[3] J. Gray and D. Siewiorek, "High-availability computer systems", *Computer*, vol. 24, no. 9, pp. 39–48, Sep. 1991, ISSN: 0018-9162. DOI: 10.1109/2.84898.

[4] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods", *ACM Comput. Surv.*, vol. 42, no. 3, 10:1–10:42, Mar. 2010, ISSN: 0360-0300. DOI: 10.1145/1670679.1670680.

[5] F. Cristian, "Understanding fault-tolerant distributed systems", *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991, ISSN: 0001-0782. DOI: 10.1145/102792.102801.

[6] I. Koren and C. M. Krishna, *Fault-tolerant systems - 1st edition*. [Online]. Available: https://www.elsevier.com/books/fault-tolerant-systems/koren/978-0-12-088525-1.

[7] A. M. Johnson and M. Malek, "Survey of software tools for evaluating reliability, availability, and serviceability", *ACM Computing Surveys*, vol. 20, no. 4, pp. 227–269, Sep. 1988, ISSN: 03600300. DOI: 10.1145/50020.50062.

[8] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, 1st ed. Cambridge University Press, 2008, ISBN: 978-0-521-87634-6.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process", *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985, ISSN: 0004-5411. DOI: 10.1145/3149.214121. [Online]. Available: http://doi.acm.org/10.1145/3149.214121.

[10] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems", *J. ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996, ISSN: 0004-5411. DOI: 10.1145/226643.226647.

[11] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony", *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988, ISSN: 00045411. DOI: 10.1145/42282.42283.

[12] P. Felber, X. Défago, R. Guerraoui, and P. Oser, "Failure detectors as first class objects", Feb. 1999, pp. 132–141, ISBN: 978-0-7695-0182-6. DOI: 10.1109/DOA.1999.794001.

[13] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1 edition. No Starch Press, Oct. 2010, ISBN: 978-1-59327-220-3.

[14] *Overview | shardingsphere*. [Online]. Available: https://shardingsphere.apache.org/document/current/en/overview/.

[15] [Online]. Available: https://proxysql.com.

[16] [Online]. Available: https://mariadb.com/kb/en/library/whitepapers/.

[17] MariaDB, "Mariadb and mysql clustering with galera - mariadb white paper", p. 5,

[18] B. Aditya and T. Juhana, "A high availability (ha) mariadb galera cluster across data center with optimized wrr scheduling algorithm of lvs - tun", in *2015 9th International Conference on Telecommunication Systems Services and Applications (TSSA)*, Nov. 2015, pp. 1–5. DOI: 10.1109/TSSA.2015.7440452.

[19] T. P. P. Council, "Tpc benchmark c (revision 5.11)", Feb. 2010.

[20] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases", *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, Dec. 2013, ISSN: 21508097. DOI: 10.14778/2732240.2732246.

[21] *Java programming language*. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/language/index.html.

[22] Gradle, "Gradle user manual: Version 5.4.1", p. 1112,

[23] *Jfrog artifactory*. [Online]. Available: https://jfrog.com/artifactory/.

[24] *Introduction | activemq artemis documentation*. [Online]. Available: https://activemq.apache.org/components/artemis/documentation/1.0.0/.

Appendix A

Configuration Files

This appendix presents all the relevant configuration files that were used on the software tools used through the lifetime of this project.

A.1 OLTPBenchmark configuration file for TPCC

```
<?xml version="1.0"?>
<parameters>

    <!-- Connection details -->
    <dbtype>mysql</dbtype>
    <driver>com.mysql.jdbc.Driver</driver>
    <DBUrl>jdbc:mysql://172.31.5.103:3307/tpcc</DBUrl>
    <username>root</username>
    <password>mypass</password>
    <isolation>TRANSACTION_READ_COMMITTED</isolation>

        <!-- Scale factor is the number of warehouses in TPCC -->
    <scalefactor>100</scalefactor>

        <!-- The workload -->
    <terminals>16</terminals>
    <works>
        <work>
            <weights>45,43,4,4,4</weights>
            <rate>10000</rate>
            <time>200</time>
            <warmup>100</warmup>
```

```
        </work>
    </works>

    <!-- TPCC specific -->
    <transactiontypes>
        <transactiontype>
            <name>NewOrder</name>
        </transactiontype>
        <transactiontype>
            <name>Payment</name>
        </transactiontype>
        <transactiontype>
            <name>OrderStatus</name>
        </transactiontype>
        <transactiontype>
            <name>Delivery</name>
        </transactiontype>
        <transactiontype>
            <name>StockLevel</name>
        </transactiontype>
    </transactiontypes>
</parameters>
```

Listing 1: OLTPBenchmark configuration file for TPCC Benchmark

A.2 Mariadb

```
[mysqld]
datadir                 = /mnt/sdb/mysql
skip-external-locking
skip_name_resolve = 1

# listen on all interfaces
bind-address                = 0.0.0.0

# InnoDB
default-storage-engine                      = InnoDB
innodb_buffer_pool_instances    = 3 # Use 1 instance per 1GB of InnoDB pool size
innodb_buffer_pool_size         = 3G # Up to 70-80% of RAM & vm.swappiness = 0
innodb_file_per_table           = 1
innodb_flush_log_at_trx_commit  = 0
innodb_flush_method             = O_DIRECT
innodb_log_buffer_size          = 16M
innodb_log_file_size            = 1G
```

```
innodb_stats_on_metadata             = 0
innodb_thread_concurrency             = 0
innodb_read_io_threads                    = 64
innodb_write_io_threads                    = 64

low_priority_updates    = 1
concurrent_insert             = 2

back_log                    = 512
thread_cache_size        = 100
thread_stack                = 192K

interactive_timeout     = 28800
wait_timeout            = 28800
connect_timeout                    = 30

# Buffer Settings
join_buffer_size            = 4M
read_buffer_size            = 3M
read_rnd_buffer_size        = 4M
sort_buffer_size            = 4M
max_heap_table_size            = 128M
tmp_table_size                   = 128M

# Fine Tuning
key_buffer_size                = 16M
max_allowed_packet        = 64M
thread_stack                = 192K
thread_cache_size   = 8

max_connections        = 1000
table_cache            = 64
thread_concurrency     = 10
thread_handling               = pool-of-threads

# Query Cache Configuration
query_cache_limit            = 1M
query_cache_size        = 16M
query_cache_type        = 1

# Prepared statements
max_prepared_stmt_count = 1048576
```

Listing 2: MariaDB server configuration file