Universidade Estadual de Campinas
Instituto de Computação

Valdomiro Luis Scannapieco Neto

Failure Detectors: Testbed and Comparative Study

Detetores de Falhas: Plataforma de Testes e Estudo Comparativo

CAMPINAS
2021

Valdomiro Luis Scannapieco Neto

Failure Detectors: Testbed and Comparative Study

Detetores de Falhas: Plataforma de Testes e Estudo Comparativo

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Luiz Eduardo Buzato

Este exemplar corresponde à versão final da Dissertação defendida por Valdomiro Luis Scannapieco Neto e orientada pelo Prof. Dr. Luiz Eduardo Buzato.

CAMPINAS
2021

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Sca63f

Scannapieco Neto, Valdomiro Luis, 1985-
Failure detectors : testbed and comparative study / Valdomiro Luis Scannapieco Neto. – Campinas, SP : [s.n.], 2021.

Orientador: Luiz Eduardo Buzato.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Detetores de falha (Computação). 2. Consenso distribuído (Computação). 3. Cluster de computadores. 4. Tolerância a falha (Computação). 5. Sistemas distribuídos. I. Buzato, Luiz Eduardo, 1961-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Detetores de falhas : plataforma de testes e estudo comparativo
Palavras-chave em inglês:
Failure detectors (Computer science)
Distributed consensus (Computer science)
Computer cluster
Fault-tolerant computing
Distributed systems
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Luiz Eduardo Buzato [Orientador]
Regina Lucia de Oliveira Moraes
Eliane Martins
Data de defesa: 21-01-2021
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-2301-116X
- Currículo Lattes do autor: http://lattes.cnpq.br/3885696710835281

Universidade Estadual de Campinas
Instituto de Computação

Valdomiro Luis Scannapieco Neto

Failure Detectors: Testbed and Comparative Study

Detetores de Falhas: Plataforma de Testes e Estudo Comparativo

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (Orientador)
  Universidade Estadual de Campinas - UNICAMP
- Profa. Dra. Regina Lucia de Oliveira Moraes
  Universidade Estadual de Campinas - UNICAMP
- Profa. Dra. Eliane Martins
  Universidade Estadual de Campinas - UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 21 de janeiro de 2021

Acknowledgements

To my family, who always supported me and helped me to reach my goals. To Prof. Dr. Luiz Eduardo Buzato for the opportunity and for all the support and participation in this work.

Resumo

Há quase trinta anos atrás, entre 1991-1992, Chandra, Toueg e Hadzilacos introduziram o conceito de detetores de falhas e mostraram como usá-los para resolver consenso em sistemas distribuídos assíncronos sujeitos a falhas parciais. Durante os anos seguintes, a abstração de detetores de falhas provou ser uma ferramenta essencial para a engenharia de sistemas distribuídos de alta disponibilidade. Resumidamente, os detetores de falhas representam uma ferramenta elegante que permite aos projetistas de sistemas distribuídos fatorar as suposições de tempo usadas para detetar falhas em algoritmos de consenso distribuído. Atualmente, um número significativo de algoritmos de deteção de falhas já foi publicado; cada um deles trazendo, supostamente, uma melhor solução para deteção de falhas; geralmente com base em uma avaliação ad hoc do algoritmo proposto. A falta de um benchmark ou plataforma de testes comum para detetores de falhas representa um obstáculo extra para engenheiros de sistemas quando eles precisam escolher um detetor de falha adequado para sua aplicação. Nesse contexto, parece razoável fazer a seguinte pergunta: qual o melhor detetor de falhas para uma determinada aplicação, executada em um determinado sistema distribuído? Neste trabalho, uma aplicação é um sistema de replicação ativa desenvolvido sobre transmissão de difusão totalmente ordenada baseada em consenso (DTOC). DTOC é o denominador comum para um grande número de aplicações reais. Chen, Toueg e Aguilera (CTA) propuseram métricas para caracterizar a qualidade do serviço prestado por um detetor de falhas. As métricas quantificam: i) quão rápido um detetor de falhas deteta falhas reais e ii) quão bem ele evita falsas deteções. Esta dissertação propõe, implementa e avalia uma plataforma de testes para detetores de falhas com base nas métricas amplamente aceitas de CTA. Em seguida, utiliza essa plataforma para buscar uma resposta para a questão feita acima. As contribuições desta pesquisa são: (i) a proposta de um método experimental para avaliar uniformemente o comportamento dos detetores de falhas, (ii) a implementação de uma plataforma de testes para apoiar o método, e (iii) um estudo comparativo de quatro detetores de falhas conhecidos.

Abstract

Almost thirty years ago, during the years of 1991-1992, Chandra, Toueg and Hadzilacos introduced the concept of failure detectors and showed how to use them to solve consensus in asynchronous distributed systems subject to partial failures. During the following years, the failure detector abstraction proved to be a key tool for the engineering of highly available distributed systems. Briefly, failure detectors represent an elegant tool that allows designers of distributed systems to factor out the timing assumptions used to detect failures in distributed consensus algorithms. Today, a substantial number of failure detection algorithms have already been published; each one of them supposedly bringing a better solution for the detection of failures; usually based on an ad hoc assessment of the failure detector proposed. The lack of a benchmark or common testbed for failure detectors represents an extra hurdle for system engineers when they have to choose a failure detector that is well suited for their application. In this context, it seems reasonable to ask the following question: what is the best failure detector for a given application, executed on a given distributed system? In this work, an application is an active replicated system developed atop a consensus-based total order broadcast (CTOB). CTOB is the common denominator for a large number of practical applications. Chen, Toueg, and Aguilera (CTA) have proposed metrics to characterize the quality of service provided by a failure detector. The metrics quantify: i) how fast a failure detector detects actual failures and ii) how well it avoids false detections. This dissertation proposes, implements and evaluates a testbed for failure detectors based on the extensively accepted CTA metrics. Then, it uses the testbed to seek an answer for the aforementioned question. The contributions of the research are: (i) the proposal of an experimental method to uniformly assess the behavior of failure detectors, (ii) the implementation of a testbed to support the method, and (iii) a comparative evaluation of four well-known failure detectors.

Glossary

- λ_M: Average mistake rate (accuracy quality of service metric).
- ACT: Aguilera et al. failure detector [14].
- CTA: Chen et al. failure detector [30].
- CTOB: Consensus-based total order broadcast.
- DTOC: "Difusão totalmente ordenada baseada em consenso".
- FAPESP: "Fundação de Amparo à Pesquisa do Estado de São Paulo".
- FD: Failure detector.
- FLP: Fischer, Lynch and Paterson [45].
- IP: Internet protocol.
- LAN: Local area network.
- LFA: Larrea et al. vanilla failure detector [59].
- LFAe: Larrea et al. epoch failure detector [73].
- LG: Load generator.
- LSD: "Laboratório de Sistemas Distribuídos".
- ms: milliseconds.
- NTP: Network Time Protocol.
- QoS: Quality of Service.
- RTT: Round-Trip Time.
- TCP: Transmission Control Protocol.
- TD_m: Detection time (speed quality of service metric).
- TE_m: Stabilization time (speed quality of service metric).
- T_G: Good period duration (accuracy quality of service metric).
- UDP: User Datagram Protocol.
- WAN: Wide area network.

Contents

Chapter 1

Introduction

Distributed consensus is a fundamental problem that is at the core of several fault-tolerant distributed applications. A system is fault-tolerant if it maintains itself available in the presence of partial component failures. Availability is only achievable via redundancy. A well-established form of redundancy is replication, that is, the maintenance of several functionally equivalent copies (replicas) of a critical component.

Unfortunately, merely having several replicas of a certain component is not enough to achieve availability. To be useful the replicas must work with each other by making each correct replica act the same. If all replicas start identical, are deterministic and go through the same computational steps, then any correct replica can accept inputs and provide the outputs. Solving replication, therefore, is reduced to feeding all replicas exactly the same inputs and in the same order. In brief, all replicas have to agree on what is the next input they are going to consume. So, agreement, or consensus, is a key algorithmic component of replication.

Suppose a set of processes that can propose values. A consensus algorithm ensures that a single one among the proposed values is chosen. Consensus is trivially solvable in the absence of failures, but is very difficult to solve in asynchronous distributed systems prone to failures. In fact, Fischer, Lynch and Paterson (FLP) [45] proved that consensus is impossible to solve in an asynchronous system - system in which there is no bound on the time it takes for either a process to execute a computation step or for a message to go from its sender to its receiver - subject to a single process failure. FLP's proof formalizes the idea that it is not possible to determine whether a process has indeed failed or is just very slow. To circumvent the FLP impossibility, Chandra and Toueg [28] created an abstraction, called failure detector (FD), which encapsulates the minimum synchrony required to achieve consensus in asynchronous distributed systems. A FD is often implemented by an object local to each process (on the same computer) that runs a failure-detection algorithm in conjunction with its counterparts at other processes. It provides some information on which processes have crashed, typically given in the form of a list of suspects, that is not always up-to-date or correct: a failure detector may take a long time to start suspecting a process that has crashed, and it may erroneously suspect a process that has not crashed (in practice this can be due to message losses and delays). It is important to note that the mistakes made by *unreliable failure detectors* should not prevent any correct process from behaving according to its specification, even

if that process is erroneously suspected to have crashed by all the other processes.

As depicted in section 3.1, Chandra and Toueg specifies FDs in terms of their eventual behavior (e.g., a process that crashes is eventually suspected). Such specifications are appropriate for asynchronous systems, in which there is no timing assumption whatsoever. However, as highlighted by Chen et al. [30], many applications have some timing constraints, and for such applications, failure detectors with eventual guarantees are not sufficient. In order to be useful, a failure detector has to be reasonably fast and accurate and that's why [30] propose a set of metrics for the Quality of Service (QoS) specification of failure detectors (section 3.2). In general, these QoS metrics should be able to describe the FD's speed (how fast it detects a failure) and its accuracy (how well it avoids mistakes).

Given that the literature offers a large set of FD algorithms, see chapter 6, it is essential to know which of them is the most suitable one for a certain application. Suitable not only in terms of speed and accuracy but also in terms of complexity of implementation, debugging and maintenance. The failure detector capable of delivering the application's required accuracy and speed for the least software engineering effort should be selected. Gumerato [49], for instance, presents a comparative study of the implementation of well-known failure detection algorithms on Local Area Network and shows no significant difference in the speed QoS of the studied algorithms. Therefore, based on this work results, the simplest algorithm could be selected. This dissertation tackles the research problem of how to select the best failure detector for building a highly available distributed application. To solve the problem a method and testbed have been proposed, developed and assessed. The assessment of the testbed is carried out by determining, out of four known failure detector algorithms, which one is the best to implement a consensus-based (Paxos) replicated application.

The contributions of this dissertation are three. Firstly, it proposes an experimental method to uniformly assess the behavior of failure detectors (see chapter 4). Secondly, based on the proposal, a testbed for failure detectors is implemented. The testbed is designed to run on a commodity computer cluster and, when assessing FDs performance, system engineers can: (i) check if the FD they want to assess is available in the testbed and, if not, implement it in conformance with the expected interface, (ii) parameterize the testbed to reproduce a specific network behavior and/or processes failures, (iii) couple the FD under assessment with the appropriate FD δ, taking into account the average latency of the network, (iv) configure a load generator and (v) run the experiments making use of an active replication toolkit that implements Paxos. The testbed could represent a step towards the increasing use of labscale environments for uniform FDs assessment, since it provides researchers with a complete framework to assess failure detectors performance. Thirdly, this work performs a comparative study of four known failure detection algorithms with different detection mechanisms (the same algorithms adopted by Gumerato): Larrea et al. [59] vanilla and epoch, Aguilera et al. [14], and Chen et al. [30]. We analyse (i) their speed metrics (detection and stabilization time) on both crash-stop and crash-recovery environments, (ii) their accuracy metrics (average mistake rate and good period duration) on a failure-free environment with the testbed emulating distinct network conditions and (iii) we assess how these different environment conditions influence consensus

using a consensus-based replication library. In terms of speed metrics, for instance, our results concluded that, differently than Gumerato, there is a FD algorithm that performs better than the others.

It is worth mentioning that, since the beginning of this work, we took a special care with two important aspects of a research: *data management* and *experiments automation* (section 5.1.3). As recognized by FAPESP [9], the appropriate *data management* facilitates the reproducibility of the results and allows the promotion of new research. In addition, it helps to carry out new analyzes, with the execution of other tests or methods of analysis. Regarding *experiments automation*, giving we dealt with a cluster of machines and several possible experiment settings, this approach was essential to the progress of the work.

Chapter 2

Distributed Systems

2.1 Definition

A distributed system is a collection of autonomous computing processes that communicate and coordinate their actions via message exchanges and seek to achieve some form of cooperation [46]. Processes are deterministic sequence of events, where an event can be either an instruction execution or the sending/receiving of a message. This definition leads to the following significant characteristics of distributed systems: concurrency of processes, lack of a global clock and autonomous failure of processes [32]. One of the most important differences between distributed and centralized systems is that the first ones may tolerate *partial failures*, that is, failures of one or more components of the system do not necessarily imply the failure of the whole system [44].

Distributed systems, hence, can be used to build highly available services. Avizienis et al. [17] define *availability* as readiness for correct service and *fault tolerance* as avoid service failures in the presence of faults. A way of building a highly available system out of less available components is to use *redundancy*, so that the system can work even when some of its parts are broken. A well-established form of redundancy is *replication*: the maintenance of copies of data at multiple computers.

A general technique for replication, that prioritizes consistency and has been widely adopted ([25], [51], [52], [63]), is *active replication* [68]. It's a method that consists in replicating data at different components and then applying the same set of operations to all replicas in the same order. More precisely, all processes sharing the same state, called replicas, behave as deterministic state machines: the transition relation is a function from (state, input) to (new state, output). Basically, several processes that start in the same state and see the same sequence of inputs will do the same thing, that is, end up in the same state and produce the same outputs. Thus, in order to implement active replication, we need to ensure that all replicas see the same inputs [58].

One of the most common ways of ensuring this is to employ *total order broadcast* [36]. Total order broadcast is a reliable broadcast communication abstraction which ensures that all processes deliver the same messages in a common global order. Sometimes it's also called atomic broadcast, because the message delivery occurs as if the broadcast were an indivisible "atomic" action: the message is delivered to all or to none of the processes and, if the message is delivered, every other message is ordered either before or after this

message [26]. It's known that total order broadcast is equivalent to *consensus*, explained in section 2.2, in *asynchronous systems* [28, 38].

An asynchronous system, which corresponds to most of the actual distributed systems, is characterized by the fact that there is no bound on the time it takes for either a process to execute a computation step or for a message to go from its sender to its receiver. That's why these systems are usually called time-free systems [66]. In a synchronous system, timing assumptions can be made and one shall assume that bounds exist. This synchronous assumption allows us to use timeouts to detect process crashes [32].

2.2 Consensus

Consensus is a fundamental problem of distributed systems that is at the core of several algorithms for fault-tolerant distributed applications such as atomic broadcast [45]. Basically, consensus allows processes to reach a common decision despite the failure of some processes [28]. In this algorithm every correct process pi proposes a value vi and all correct processes have to decide on some value v, in relation to the proposed values [66]. The requirements of a regular consensus algorithm are that the following conditions should hold for every execution of it [26]:

- *Termination*: Every correct process eventually decides on some value.
- *Validity*: If a process decides on v, then v was proposed by some process.
- *Integrity*: No process decides twice.
- *Agreement*: No two correct processes decide differently.

As highlighted by Lampson [58], consensus has more applications than only general replicated state machines. It can also be used (i) in distributed transactions where all the processes need to agree on whether a transaction commits or aborts and (ii) to elect a leader of a group of processes without knowing exactly what the members are.

2.3 Impossibility of Consensus

Fischer, Lynch and Paterson (FLP) [45] prove the impossibility of solving both consensus and atomic broadcast deterministically in an asynchronous system subject to even a single, unannounced, process failure. Their proof involves showing that there is always some continuation of the processes' execution that avoids consensus being reached [32]. FLP's proof formalizes the idea that it's not possible to determine whether a process has indeed crashed or is just very slow.

To circumvent the FLP impossibility, previous research focused on the use of randomisation techniques [31], on the definition of some weaker problems and their solutions [16, 22, 24, 39] or on the study of several models of partial synchrony [38, 40]. Nevertheless, the impossibility of deterministic solutions to many agreement problems remains

a major obstacle to the use of the asynchronous model of computation for fault-tolerant distributed computing [28].

Chandra and Toueg [28] propose an alternative approach to work around FLP and to broaden the applicability of the asynchronous model of computation. Since this impossibility result stem from the inherent difficulty of determining whether a process has actually crashed or is very slow, Chandra and Toueg propose to augment the asynchronous model of computation with a model of an external failure detection mechanism that can make mistakes. In particular, they introduce the concept of *unreliable failure detectors* for systems with crash failures.

Failure detectors are the focus of this dissertation and will be covered in detail in the next chapter.

Chapter 3

Failure detectors

3.1 Definition

A failure detector (FD) is a basic building block of distributed systems that are designed to provide reliable and continuous services despite the failures of some of their components [30]. The failure detector abstraction is an elegant way to solve consensus in a modular manner by defining high-level properties that encapsulate synchrony assumptions [41]. A FD can be seen as a distributed oracle that signals the ocurrence of processes' failures and is often implemented by an object local to each process (on the same computer) that runs a failure-detection algorithm in conjunction with its counterparts at other processes.

A failure detector can be either **reliable** or **unreliable**. A *reliable failure detector*, which requires a synchronous system, is always accurate in detecting a process's failure whereas an *unreliable failure detector*, which does not require a synchronous system, is not necessarily accurate. *Reliable failure detectors* answer processes' queries with either *Unsuspected* or *Failed*, the latter meaning that the monitored process has indeed crashed, whilst *unreliable failure detectors* answer queries with either *Unsuspected* or *Suspected*. These results returned by *unreliable failure detectors* are hints, that may or may not accurately reflect whether the process has actually failed [32]. It is important to note that the mistakes made by *unreliable failure detectors* should not prevent any correct process from behaving according to its specification, even if that process is erroneously suspected to have crashed by all the other processes.

Chandra and Toueg [28] analyze the properties that a failure detector must have in order to solve consensus and atomic broadcast in an asynchronous system. Classes of FDs are formalized and classified according to the properties of **completeness** and **accuracy**. **Completeness** requires that a failure detector eventually suspects every process that actually crashed, whereas **accuracy** restricts the mistakes that a failure detector can make. There are two degrees of completeness and four degrees of accuracy, resulting in eight classes of FDs as shown in Figure 3.1 and listed below:

- *Strong Completeness*: eventually every process that crashes is permanently suspected by every correct process.
- *Weak Completeness*: eventually every process that crashes is permanently suspected by some correct process.

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\Diamond\mathcal{P}$	*Eventually Strong* $\Diamond\mathcal{S}$
Weak	$\mathcal{L}$	*Weak* $\mathcal{W}$	$\Diamond\mathcal{L}$	*Eventually Weak* $\Diamond\mathcal{W}$

Figure 3.1: Eight classes of failure detectors defined in terms of accuracy and completeness [28].

$$\mathcal{P} \cong \mathcal{L} \quad , \quad \mathcal{S} \cong \mathcal{W} \quad , \quad \Diamond\mathcal{P} \cong \Diamond\mathcal{L} \quad \text{e} \quad \Diamond\mathcal{S} \cong \Diamond\mathcal{W}$$

Figure 3.2: Failure detectors equivalence [28].

- *Strong Accuracy*: no process is suspected before it crashes.
- *Weak Accuracy*: some correct process is never suspected.
- *Eventual Strong Accuracy*: there is a time after which correct processes are not suspected by any correct process.
- *Eventual Weak Accuracy*: there is a time after which some correct process is never suspected by any correct process.

Regarding the accuracy properties, given that even the *Weak Accuracy* could be difficult to achieve (at least one correct process is never suspected), two weaker properties were defined, requiring that strong accuracy or weak accuracy are only eventually satisfied.

Chandra and Toueg introduce the concept of reducibility among failure detectors: a FD a is reducible to a FD b if there is a distributed algorithm that can transform b into a. Basically, two failure detectors are equivalent if they are reducible to each other. By employing the concept of reducibility, the equivalence shown in Figure 3.2 was proved, reducing the eight classes of failure detectors to four (Figure 3.3) and identifying how consensus could be solved for each one.

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\Diamond\mathcal{P}$	*Eventually Strong* $\Diamond\mathcal{S}$

Figure 3.3: Four classes of failure detectors after reducibility [28].

In [27], Chandra et al. proves that $\diamond W$ is the weakest failure detector that can be used to solve consensus in asynchronous systems with a majority of correct processes. In the same work, it's defined the class Ω, equivalent to $\diamond W$, that acts as a leader elector by providing a correct process as output. This class is essential to this work because the developed testbed is built using Treplica [73], an active replication toolkit, that implements Paxos [56] as consensus algorithm and whose liveness is determined by Ω. The failure detectors studied in this dissertation are detailed in section 3.3.

3.2 Quality of service

Chandra and Toueg specifies FDs in terms of their eventual behavior (e.g., a process that crashes is eventually suspected). Such specifications are appropriate for asynchronous systems, in which there is no timing assumption whatsoever. However, as highlighted by Chen et al. [30], many applications have some timing constraints, and for such applications, failure detectors with eventual guarantees are not sufficient. In order to solve consensus, for instance, it's expected that the Ω associated with each replica will indicate the leader for periods long enough for the resolution of consensus instances [58]. If consensus cannot progress then the application built upon it will not progress as well. In this work, we say that a FD (or leader elector) *stabilizes* when it indicates a leader for a long enough period of time to guarantee consensus's liveness.

Hence, in order to be useful, a failure detector has to be reasonably fast and accurate and that's why Chen et al. [30] propose a set of metrics for the QoS specification of failure detectors. In general, these QoS metrics should be able to describe the FD's speed (how fast it detects a failure) and its accuracy (how well it avoids mistakes). Three primary metrics were proposed [30] for the QoS of failure detectors:

- T_D (detection time): the time interval between the crash of the process and the time in which the failure detector starts to suspect the process in a permanent way; this metric quantifies the delay of the failure detector.
- T_M (mistake duration): the time the failure detector takes to correct a mistake; it measures the time interval between an erroneous detection and its correction.
- T_{MR} (mistake recurrence time): the time between two successive mistakes.

As it can be noted, T_D quantifies the **completeness**, whereas T_M and T_{MR} quantify the **accuracy** of a failure detector. Figures 3.4 and 3.5 illustrate the operational meaning of the primary metrics. Consider a system of two processes p and q connected by a lossy communication link and suppose that the failure detector at q monitors process p and that q does not crash. Figure 3.4 shows how T_D is computed by analysing when process p crashes and when process q starts to suspect p permanently. Figure 3.5, on the other hand, exemplifies T_M and T_{MR}. In this case process p never crashes, however, q can make mistakes indicating p as suspected. Therefore, the output of the failure detector at q is either S, "I suspect that p has crashed", or T, "I trust that p is up". A transition occurs

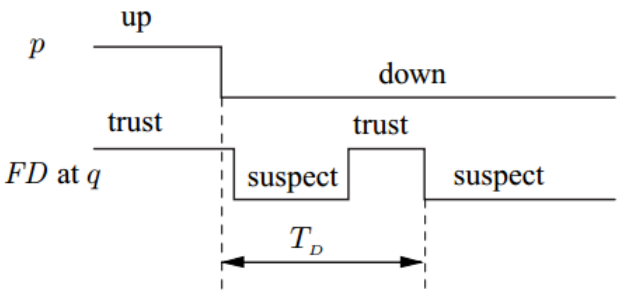

Figure 3.4: Detection time T_D [30].

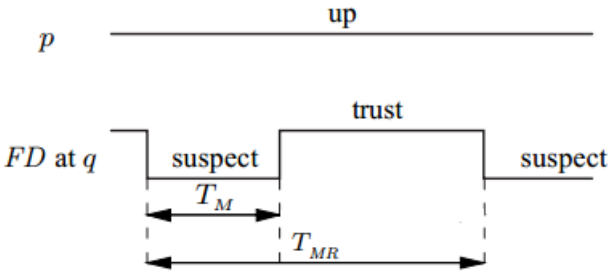

Figure 3.5: Mistake duration T_M and mistake recurrence time T_{MR} [30].

when the output of the FD at q changes: a *S-transition* occurs when the output at q changes from T to S; a *T-transition* occurs when the output at q changes from S to T.

In addition to the primary accuracy metrics T_M and T_{MR}, Chen et al. [30] also propose four derived metrics:

- λ_M (average mistake rate): measures the rate at which a failure detector make mistakes, i.e., it is the average number of *S-transitions* per time unit. This metric is important to long-lived applications where each failure detector mistake (each *S-transition*) results in a costly interrupt.

- P_A (query accuracy probability): probability that the failure detector's output is correct at a random time. This metric is important to applications that interact with the failure detector by querying it at random times. A Treplica process, for instance, queries its FD periodically.

- T_G (good period duration): measures the length of a good period. More precisely, T_G is a random variable representing the time that elapses from a *T-transition* to the next *S-transition*. Many applications can make progress only during good periods - periods in which the failure detector makes no mistakes.

- T_{FG} (forward good period duration): this is a random variable representing the time that elapses from a random time at which q trusts p, to the time of the next *S-transition*. This random variable is a predictor of future behavior and, as in our study we are mainly interested on what actually has happened, we have decided not to compute it.

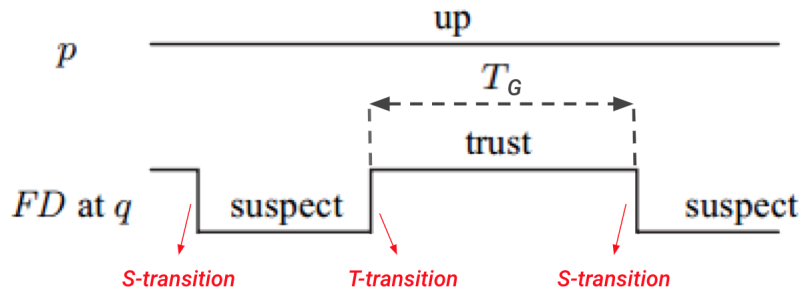

Figure 3.6: λ_M (average mistake rate) and T_G (good period duration) [30].

Two of these metrics, λ_M and T_G, are very relevant to the current work specially when assessing the impact of FD's accuracy on consensus. Consensus-based applications, like the replication library adopted to assess consensus performance in this work (Treplica [73]), may rely on good periods to make progress and failure detector mistakes can be costly. Figure 3.6 illustrates how both metrics work: T_G indicates the time that elapses from a *T-transition* to the next *S-transition* while λ_M is the average number of *S-transitions* per time unit.

In large scale systems, maintaining QoS guarantees for failure detectors is a challenging task due to size and geographical scalability [65]. In some situations, we cannot assume that the probabilistic behavior of the network doesn't change. For instance, a corporate network may have one behavior during working hours and a completely different one during lunch time or at night. Such networks require a failure detector that adapts to the changing conditions, i.e., that dynamically reconfigures itself to meet some given QoS requirements [30].

3.3 Failure detectors studied

This work assess four failure detectors (the same algorithms adopted by Gumerato [49]): Larrea et al. [59] vanilla (LFA) and epoch (LFAe), Aguilera et al. [14] (ACT) and Chen et al. [30] (CTA).

These algorithms were selected due to the fact that each of them focus on a different failure detection strategy as described in the next sections. Besides that, originally, the chosen algorithms belong to different classes of FDs as defined by Chandra and Toueg [28] and mentioned in section 3.1. Larrea et al. [59] and Chen et al. [30] belong to Ω, while Aguilera et al. [14] originally belongs to $\diamond$S, although it was adapted to be an Ω in this work.

3.3.1 Larrea et al. vanilla (LFA)

Larrea et al. vanilla (LFA) [59], algorithm 1, is the simplest algorithm studied. LFA uses a mechanism of message broadcast and a mechanism of local timing that do not depend upon any global clock synchronization mechanism. The goal of this FD is to return to the upper-layer components a trusted process, by definition, the process with

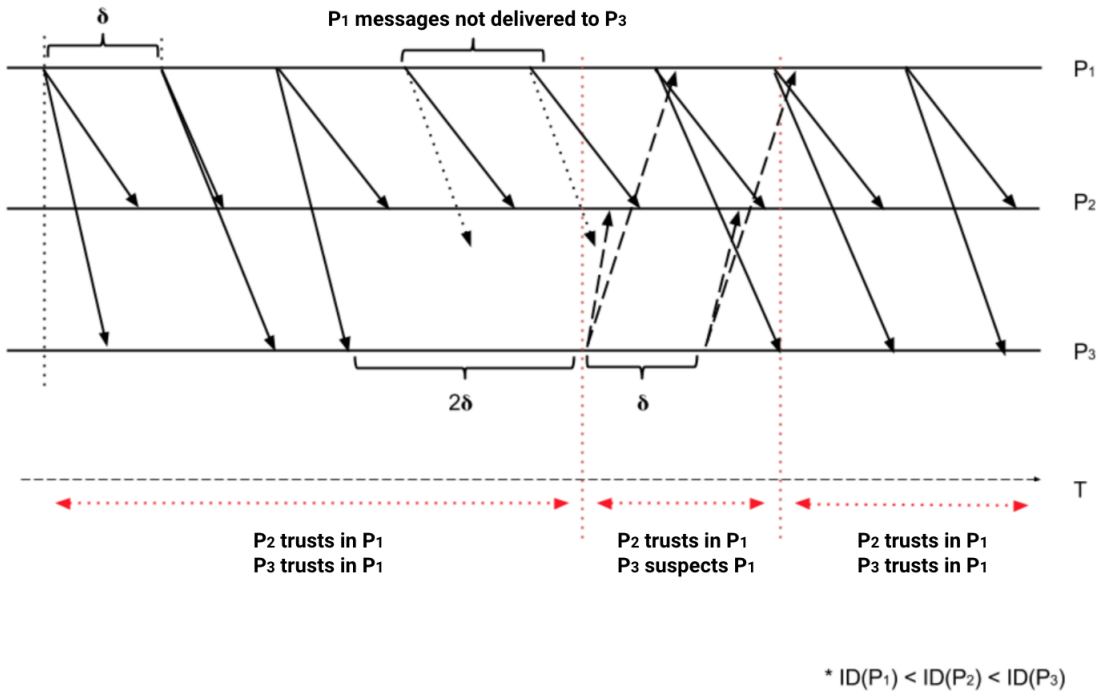

Figure 3.7: LFA timing schema [49].

the lowest identifier. Each process has a unique system identifier, its *id* (line 2), which can be ordered in a global and non-decreasing order. During the system initialization (line 3), each process considers itself the system's leader (lines 4 and 5) and starts to send messages periodically to the other processes (line 17). When a process p receives a message from a process q, whose *id* is lower than p's *id* (line 8), p automatically considers q as the leader (line 9). All processes follow the same logic and, this way, the process with the lowest identifier will stabilize as the leader. After the election of a process q as the leader, only q keeps sending periodical heartbeats (line 17). If a process p stops receiving messages from the current leader, a new leader election process takes place, with p attempting to be elected (lines 19 and 20).

Figure 3.7 illustrates this dynamic. The leader *P1* sends periodical heartbeats to processes *P2* and *P3* every δ units of time. At a certain moment, *P3* stops receiving *P1* messages due to, for instance, a temporary network issue. After at most 2δ units of time since the last *P1* message received, *P3* considers itself as leader and starts sending heartbeats to the other processes in an attempt to be elected as the system leader. Eventually, *P1* messages are delivered to *P3* again that, in turn, recognizes *P1* as leader and stops broadcasting messages. The single timing parameter δ must be chosen carefully in order to be a FD, at the same time, reasonably fast and accurate as portrayed in the QoS section 3.2.

LFA presents message exchanges complexity of O(n), when a stable leader in is place, and $O(n^2)$ during a leader election process.

Algorithm 1 Larrea et al. vanilla (LFA)

```
1:  local process p variables
2:      id, δ, leader, leaderTimestamp
3:  upon initialization:
4:      leader ← id
5:      leaderTimestamp ← now
6:      set timeout with δ time units
7:  upon receive message m_q:
8:      if m_q.id < leader then
9:          leader ← m_q.id
10:     endif
11:     if m_q.id = leader then
12:         leaderTimestamp ← now
13:     endif
14: upon timeout δ:
15:     set timeout with δ time units
16:     if id = leader then
17:         broadcast message m_p = {id}
18:     else if (now - leaderTimestamp) > δ then
19:         leader ← id
20:         broadcast message m_p = {id}
21:     endif
```

3.3.2 Larrea et al. epoch (LFAe)

This variation of the LFA algorithm, algorithm 2, was designed, implemented and used in the work that presents Treplica [73], an active replication toolkit that implements Paxos [56] as consensus algorithm. The motivation for creating a new FD came from the fact that LFA vanilla doesn't apply any stability mechanism for the elected leader, so that any process that suspects it could initiate a new election process. That behavior is not desirable because leader changes are costly and must be avoided whenever possible.

In order to avoid unnecessary leader changes a mechanism of epoch numbers was introduced (lines 6 and 7) to work along with the timing one inherited from LFA. This mechanism makes use of local non-decreasing counters in each process, that indicate the epoch of the current process (myEpoch) and the epoch of the leader (leaderEpoch). In case of a temporary communication failure among the leader and the other processes, the leader keeps increasing its epoch number (line 21) since messages are still being sent (line 23). This way, when the communication is reestablished, the original leader is reelected because it has the highest epoch number. In LFAe, therefore, the elected leader is either the process with the highest epoch number or, in case of more than one replica with the very same epoch, the one with the lowest identifier.

LFAe, like LFA, presents message exchanges complexity of O(n), when a stable leader in is place, and $O(n^2)$ during a leader election process.

Algorithm 2 Larrea et al. epoch (LFAe)

1: **local process** p **variables**
2: id, δ, myEpoch, leader, leaderTimestamp, leaderEpoch
3: **upon** initialization:
4: leader $\leftarrow$ id
5: leaderTimestamp $\leftarrow$ now
6: leaderEpoch $\leftarrow$ 0
7: myEpoch $\leftarrow$ 0
8: set timeout with δ time units
9: **upon** receive message m_q:
10: **if** (m_q.epoch $>$ leaderEpoch $\wedge$ m_q.id $\neq$ leader) $\vee$
11: (m_q.epoch = leaderEpoch $\wedge$ m_q.id $<$ leader) **then**
12: leader $\leftarrow$ m_q.id
13: **endif**
14: **if** m_q.id = leader **then**
15: leaderEpoch $\leftarrow$ m_q.epoch
16: leaderTimestamp $\leftarrow$ now
17: **endif**
18: **upon** timeout δ:
19: set timeout with δ time units
20: **if** id = leader **then**
21: myEpoch++
22: leaderEpoch $\leftarrow$ myEpoch
23: broadcast message $m_p = \{id, myEpoch\}$
24: **else if** (now - leaderTimestamp) $> \delta$ **then**
25: leaderEpoch $\leftarrow$ myEpoch
26: leader $\leftarrow$ id
27: broadcast message $m_p = \{id, myEpoch\}$
28: **endif**

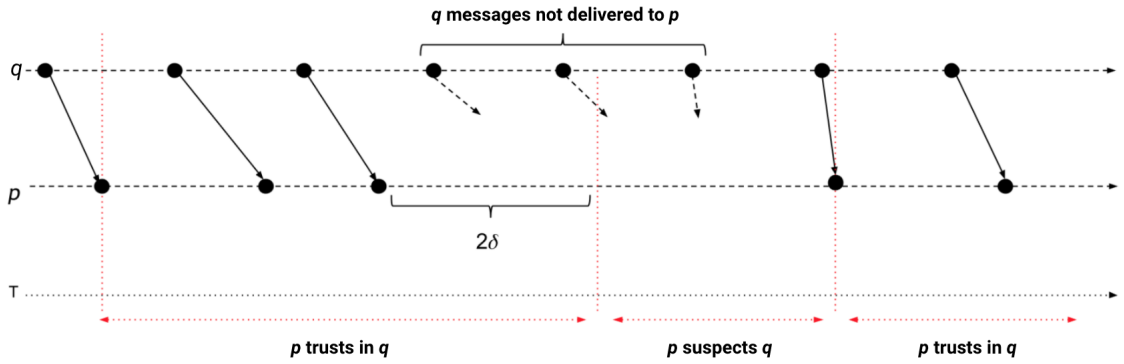

Figure 3.8: ACT timing schema [49].

3.3.3 Aguilera et al. (ACT)

Aguilera et al. [14], algorithm 3, is an algorithm based on a mechanism of periodical heartbeats. Each process has a unique system identifier (line 2) which can be ordered in a non-decreasing way. As output, the algorithm provides a vector (line 3) with the number of heartbeats received from each process of the system (line 13). The authors don't specify any timing mechanism in the original article so that the algorithm doesn't determine which processes failed or not. For this reason, ACT was modified in [49] to turn it into a leader elector and have its performance assessed when coupled to our active replication toolkit that implements Paxos and whose liveness is determined by an Ω FD. Basically, a timing mechanism was introduced (line 17) so that the FD algorithm could point out if a process is correct (process p received a message from process q in less than δ time units ago) or not and the leader among them (process with the lowest identifier, line 27).

Figure 3.8 presents how ACT works when some messages aren't delivered, leading a process p to suspect a process q temporarily. The messages are exchanged periodically between p and q (although the image only shows messages sent from q to p) with the failure detection triggered after, at most, 2δ cycles.

ACT presents message exchanges complexity of $O(n^2)$ during its whole execution because all processes keep broadcasting messages to the other processes even with a stable leader elected.

3.3.4 Chen et al. (CTA)

Chen et al. [30] propose a new FD considering two undesirable characteristics of common failure detection algorithms. FDs commonly used in practice (like LFA) work as follows: at regular time intervals, process p sends a heartbeat message to q; when q receives a heartbeat message it trusts p and starts a timer with a fixed timeout value δ; if the timer expires before q receives a newer heartbeat message from p, then q starts suspecting p. This algorithm has two undesirable characteristics:

1. There is a dependency on past heartbeats: the timer for message *mi*, with *mi* being the i-th heartbeat, is started upon the receipt of *mi-1*, and so if *mi-1* is "fast", the

Algorithm 3 Aguilera et al. (ACT)

1: **local process** p **variables**
2: id, δ, η, leader
3: V_i: $\{id, counter, previousCounter, trustable\}$ $\forall_i \in p$
4: **upon** initialization:
5: leader $\leftarrow$ id
6: **for each** V_i:
7: V_i.counter $\leftarrow 0$
8: V_i.previousCounter $\leftarrow 0$
9: V_i.trustable $\leftarrow$ false
10: set timeout with δ time units
11: set timeout with η time units
12: **upon** receive message m_q:
13: V_q.counter++
14: **upon** timeout η:
15: set timeout with η time units
16: broadcast message $m_p = \{id\}$
17: **upon** timeout δ:
18: set timeout with δ time units
19: **for each** V_i:
20: **if** (V_i.counter - V_i.previousCounter) > 0 **then**
21: V_i.trustable $\leftarrow$ true
22: **else**
23: V_i.trustable $\leftarrow$ false
24: **endif**
25: V_i.previousCounter $\leftarrow$ V_i.counter
26: leader $\leftarrow$ V_q.id **where**
27: V_q.trustable = true $\wedge$ $\{\nexists r \rightarrow (Vr.id < Vq.id \wedge Vr.trustable = true)\}$

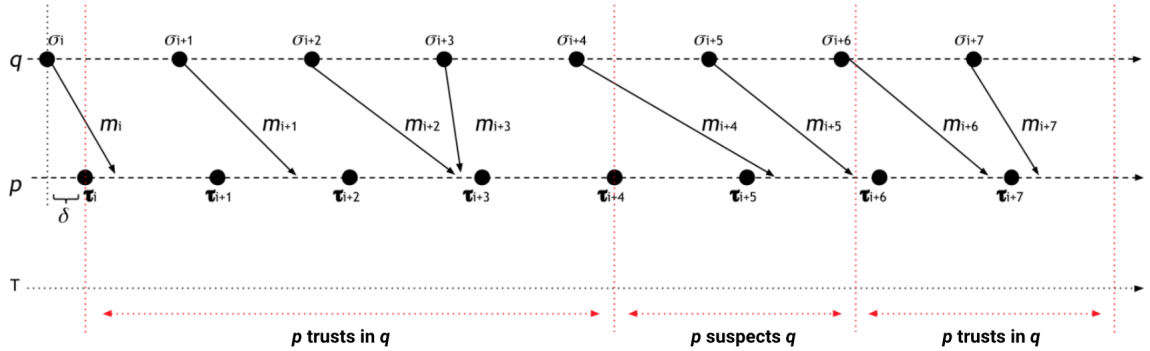

Figure 3.9: CTA timing schema [49].

timer for *mi* starts early and this increases the probability of a premature timeout on *mi*.

2. Suppose p sends a heartbeat just before it crashes, and let d be the delay of this last heartbeat. In the commonly used algorithms, q would permanently suspect p only $d + \delta$ time units after p crashes. Thus, the worstcase detection time for this algorithm is the maximum message delay plus δ. This is impractical because in many systems the maximum message delay is orders of magnitude larger than the average message delay.

As can be noted the source of the above problems is that even though the heartbeats are sent at regular intervals, the timers to "catch" them expire at irregular times and the algorithm proposed by Chen et al. eliminates this problem.

Chen et al. FD, algorithm 4, is also based on periodic messages at fixed intervals, however it uses a mechanism of pre-defined timing windows to receive messages and to detect whether a process is correct or not. The system has a fixed number n of processes and each of them sends indexed messages *mi* to the other processes (line 21). Each process of the system has a unique *id* which can be ordered in a non-decreasing way (line 2). A process p considers a process q correct in the interval [*ti*, *ti+1*] if the message *mi* sent by q is received within the interval [*ti*, *ti+1*] (lines 16 and 17). The timing windows are built locally on each replica and must be synchronized accordingly so that the failure detection/leader election is coherent in the entire system.

Figure 3.9 illustrates how Chen et al. (CTA) works. The processes keep a local clock that always advances, used to calculate the moments of sending messages and intervals [*ti*, *ti+1*]. Upon receiving a message *mi* from process q, process p checks if it was received within the time interval [*ti*, *ti+1*]. If so, the process q is considered correct by the process p; otherwise, it's considered as a failure. The leader is chosen as the process with the lowest id among the correct processes.

CTA also presents message exchanges complexity of $O(n^2)$ during its whole execution since all processes broadcast messages to the other processes even with a leader elected.

Algorithm 4 Chen et al. (CTA)

1: **local process** p **variables**
2: id, δ, η, sequenceNumber, leader
3: V_i: $\{id, sequenceNumber, \tau i, trustable\}$ $\forall_i \in p$
4: **upon** initialization:
5: leader $\leftarrow$ id
6: sequenceNumber $\leftarrow 0$
7: **for each** V_i:
8: V_i.sequenceNumber $\leftarrow 0$
9: $V_i.\tau_i \leftarrow$ now
10: V_i.trustable $\leftarrow$ false
11: set timeout with δ time units
12: set timeout with η time units
13: **upon** receive message m_q:
14: update $V_q.\tau_i$
15: V_q.sequenceNumber $\leftarrow$ m_q.sequenceNumber
16: **if** now $\in V_q.\tau_i$ **then**
17: V_q.trustable $\leftarrow$ true
18: **endif**
19: **upon** timeout η:
20: set timeout with η time units
21: broadcast message $m_p = \{id, sequenceNumber++\}$
22: **upon** timeout δ:
23: set timeout with δ time units
24: **for each** V_i:
25: **if** now $\notin V_i.\tau_i$ **then**
26: V_i.trustable $\leftarrow$ false
27: **endif**
28: leader $\leftarrow$ V_q.id **where**
29: V_q.trustable = true $\wedge$ $\{\nexists r \rightarrow (Vr.id < Vq.id \wedge Vr.trustable = true)\}$

3.3.5 Algorithms complexity

In terms of complexity of implementing, troubleshooting and maintaining each failure detector algorithm, the order is the following as detailed in [49]:

$$\text{LFA} < \text{LFAe} < \text{ACT} < \text{CTA}$$

with LFA being the simplest FD whereas CTA is the most complex one, as demonstrated in their corresponding pseudocodes.

Algorithm	**Original Class**	**Implementation Class**	**Message Exchanges Complexity**
LFA	Ω	Ω	$O(n)$
LFAe	Ω	Ω	$O(n)$
ACT	$\diamond$S	Ω	$O(n^2)$
CTA	Ω	Ω	$O(n^2)$

Table 3.1: FD Properties [49].

Table 3.1 presents the main properties of the studied failure detectors. Originally, ACT doesn't belong to Ω class, however, as described in section 3.3.3 a modification was introduced to turn it into a leader elector. Therefore, all FD algorithms belong to the Ω class. Regarding message exchanges complexity, LFA and LFAe present linear complexity when a stable leader is in place, since only it keeps sending messages to the other replicas. ACT and CTA, on the other hand, present quadratic complexity, since messages are exchanged among all processes during the whole algorithm execution.

Chapter 4

A Testbed for Failure Detectors

Given that the literature offers a large set of FD algorithms (chapter 6), it's essential to know which of them is the most suitable one for a certain application. Nonetheless, the lack of a benchmark or testbed for failure detectors represents an extra hurdle for system engineers when they have to choose a FD that is well suited for their application. In order to support system engineers in their quest for a suitable FD algorithm for their fault-tolerant system, this dissertation proposes and implements a testbed for failure detectors. The testbed is designed to run on a commodity computer cluster and the procedure an engineer has to follow to test failure detectors is comprised of the following steps:

1. Check if the FD the engineer wants to assess is available in the testbed. If yes, check the current implementation. Otherwise, implement it in conformance with the testbed failure detector programming interface.

2. Parameterize the testbed to reproduce a specific network behavior and/or processes failures.

3. Couple the FD under assessment with the appropriate FD δ, taking into account the average latency of the network.

4. Configure a load generator.

5. Run the experiments making use of an active replication toolkit that implements Paxos.

The testbed could represent a step towards the increasing use of labscale environments for uniform FDs assessment, since it provides researchers with a complete framework to assess failure detectors performance. Figure 4.1 presents a high-level software architecture of the testbed. The software architecture comprises four major parts:

- In red: an application responsible for **load generation** making use of a Replicated HashMap.

- In yellow: an **active replication toolkit**, Treplica [73], that implements Paxos and Fast Paxos with its main components represented: Replicated State Machine and Distributed Consensus.

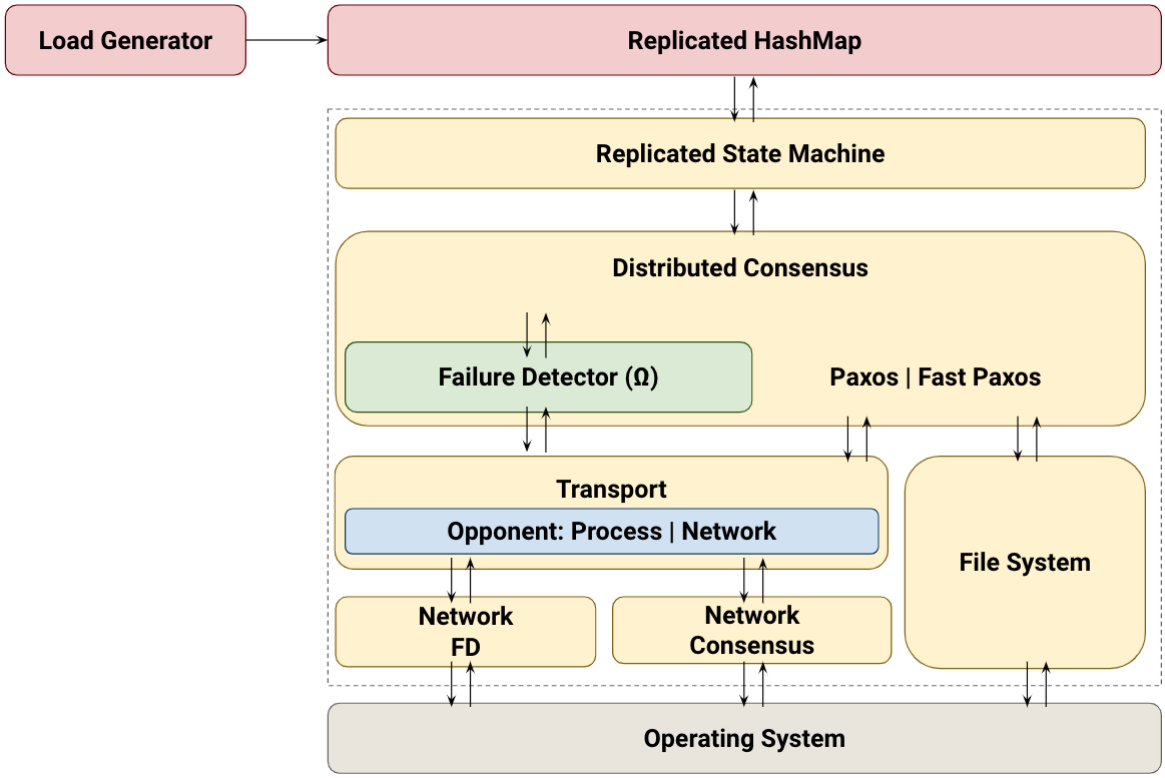

Figure 4.1: Testbed software architecture.

- In green: a **failure detector** (explained in 3.1).
- In blue: a process and network ***opponent*** running on the transport layer.

Next, each of the main components is described in further details.

4.1 Load generator

The load generator application (LG), represented in red in Figure 4.1, instantiates a Replicated HashMap that uses the state machine's interface from Treplica to replicate the generated operations among a couple of replicas. In terms of workload, the LG only performs write operations by attempting to add, per second, a specific number of key:value pairs to the map via method *put(key: string, value: string)*. We mention "attempt to add" because all write operations on the Replicated HashMap are blocking and only return when they indeed complete. The number of puts per second is defined in a config file and the actual value during the experiments of this dissertation, along with the number of replicas, is depicted in section 5.2.

It's important to highlight that all replicas are responsible for executing the workload over the HashMap. Therefore, if we have X replicas and Y puts/second configured, each of the X replicas will keep calling put during the experiments in an attempt to meet the Y write rate.

Finally, the data pattern applied in this work is from applications that implement distributed consensus to solve data replication. Treplica [73] is adopted in this project, nonetheless, other similar applications can be found in the literature such as Google Chubby [25], Amazon Dynamo [35] and Apache Zookeeper [3].

4.2 Treplica

Treplica [73], represented in yellow in Figure 4.1, is an active replication library developed atop a consensus-based total order broadcast. It simplifies the development of high-available applications by making transparent the complexities of dealing with replication and persistence. Treplica implements at its core a replication protocol that gives applications the ability of tolerating crashes and recoveries of a subset of their components without having to worry about the consistency of the replicated data. The tool proposes the idea of handling and presenting to the application programmer a unified programming abstraction for replication and persistence. It proposes the use of consensus as a foundation for construction of such unified replication tool. Treplica implements two consensus algorithms, Paxos [56] and Fast Paxos [57], although only the first one was used in this current work since Fast Paxos does not depend on a coordinator (Ω) as Paxos does.

Figure 4.1 presents a high-level view of Treplica's components:

- *Replicated State Machine*: interface used to program the routines of Treplica's applications. The distributed HashMap is implemented using this interface.
- *Distributed Consensus*: considered Treplica's core, implements the functions of ordering the State Machine's operations execution.
- *File System*: used to store all Treplica's persistent data. This component keeps the data saved even in case of a process crash.
- *Transport / Network*: transport layer used by Treplica to exchange messages with other processes. UDP is the transport layer protocol adopted.

The testbed architecture follows Gumerato's [49] important discoveries and adopts a dedicated physical network for Treplica, isolated from the failure detector's network, so that there is no interference between the algorithms since both rely on message exchanges among the replicas.

4.3 Failure detector

Four failure detectors were implemented and compared making use of the testbed: Larrea et al. [59] vanilla and epoch, Aguilera et al. [14] and Chen et al. [30]. Other FDs can be implemented and coupled to Treplica, in our testbed, by implementing the *IFailureDetector* interface provided which requires two straightforward functions:

1. Return the current leader by implementing the **int getLeader()** method. The integer to be returned indicates the leader process unique system identifier.

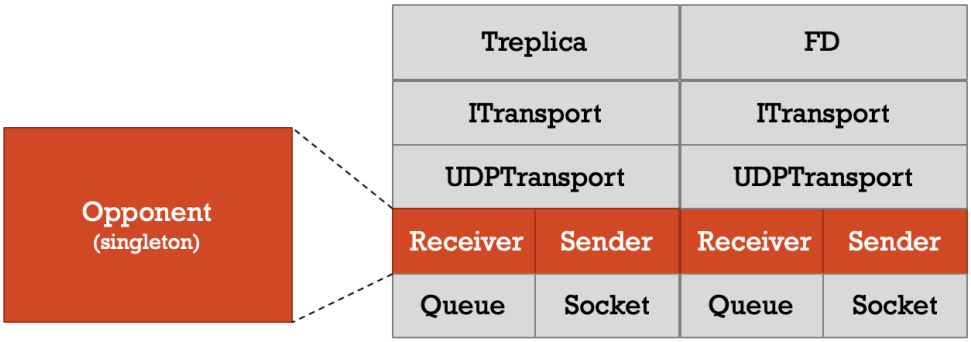

Figure 4.2: Opponent's software architecture.

2. Call a callback method, **void leaderChange(int newLeader)**, whenever the leader changes. The parameter expected is also the leader process identifier.

With this two functions the upper-layer algorithms that rely on failure detectors building blocks can obtain the identifier of the current leader proactively and reactively and progress accordingly.

In this work all FDs were developed in Java and, the choice of using this language, was oriented by the fact that Treplica is also written in Java. By using the same language, the task of coupling the component is simplified and, moreover, it makes possible the reuse of components such as the transport layer used by both Treplica and the failure detectors.

4.4 Opponent

An *opponent*, represented in blue in Figure 4.1, was proposed and developed to provide a way of abstracting (i) the inherent behavior of diverse networks and (ii) processes failures. A modification on Treplica's transport layer was performed to couple the *opponent* as depicted in Figure 4.2. The *opponent* was implemented as a single instance responsible for dictating three possible actions to be performed over each sending/receiving packet:

- *Pass*: No action is performed and the packet is sent or received without any interference.
- *Drop*: The packet is discarded.
- *Delay*: A delay is applied, asynchronously by dedicated threads, to the packet before sending or receiving it.

In order to abstract a process failure, the *opponent* has a mechanism that emulates a total failure in the process communication by dropping all the packets sent/received by the failing process. On the other hand, to reproduce distinct network behaviors, the selection of which action to be applied follows a probability distribution. In this work, the library *Apache Commons Mathematics* [2] was imported to provide commonly used distributions [13] and its *EnumeratedIntegerDistribution* and *NormalDistribution* classes were the ones

adopted in our experiments. In the *EnumeratedIntegerDistribution* [5], an equiprobable distribution, values are assigned probability based on their frequency. For example, [0, 1, 1, 2] as input creates a distribution with 25% of chance to return the value 0, 50% of chance to return 1 and 25% of chance to return 2. In the *NormalDistribution* [7], or Gaussian distribution, we define a mean (expectation of the distribution) and a standard deviation, leading to a curved flaring shape. In our work negative values mean *drop*, zeroed values mean *pass* and positive values represent the amount of *delay* in milliseconds.

This *opponent* can be tuned to reproduce different network behaviors. Other distributions other than the ones adopted in this work, for instance *PoissonDistribution* [8] or *ExponentialDistribution* [6], can be easily coupled to the experimental environment by implementing the *OpponentStrategy* interface provided.

Chapter 5

Comparative Failure Detection

This dissertation, performs a comparative study of four known failure detection algorithms with different detection mechanisms: Larrea et al. [59] vanilla and epoch, Aguilera et al. [14], and Chen et al. [30]. We analyse (i) their speed metrics (detection and stabilization time) on both crash-stop and crash-recovery environments, (ii) their accuracy metrics (average mistake rate and good period duration) on a failure-free environment with the testbed emulating distinct network conditions and (iii) we assess how these different environment conditions influence consensus using a replication library that implements Paxos.

The next sections will cover materials and methods, the experiments performed and their respective results.

5.1 Materials and methods

5.1.1 Platform

All experiments were executed in the LSD laboratory ("Laboratório de Sistemas Distribuídos") from IC/UNICAMP. The testbed environment consisted of a cluster of 5 machines, each one equipped with two Intel-Xeon quad-core processors of 2.40 GHz and 12 GB of memory RAM. Two switches 3Com 4200G Gigabit Ethernet with 24 ports and Round-Trip Time (RTT) of less than 1 ms connected the machines. All machines run Gnu/Linux Debian 6.0 with kernel Linux 2.6.32 SMP 64-bit as operating system and a Java virtual machine OpenJDK version 1.8.

Clock synchronization

The clock synchronization among the machines was performed through NTP (Network Time Protocol) [64], a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. NTP is built over UDP/IP protocols and uses a probabilistic mechanism to implement synchronization. This mechanism is based on message exchanges among synchronization computers distributed in different hierarchy levels. The 64-bit timestamps used by NTP consist of a 32-bit part for seconds and a 32-bit part for fractional second, giving a theoretical resolution of 2^{-32} seconds (233 picoseconds).

The clock precision in the cluster, estimated by the NTP protocol itself, was of $4.76\text{x}10^{-7}$ s, whereas the message exchanges among the machines used to occur in about 1 ms (10^{-3} s). For this reason, when assessing the experiments output, we considered that the clocks of all machines present in the cluster were synchronized [49].

5.1.2 Software

The implemented testbed for failure detectors is adopted to run the experiments. The setup was the following:

- Load generator: it uses a Replicated HashMap with 5 replicas; each replica calling 500 puts per second. Each experiment consisted of 60 executions/iterations of 600 seconds each. The load generator started at 40 seconds, after 30 seconds of warm up.
- Treplica: the consensus algorithm adopted is Paxos, also with 5 replicas. It follows Gumerato's [49] discoveries by using a dedicated physical network, isolated from the FD's network, in order to avoid interference between the consensus and failure detector algorithms given both rely on message exchanges among the replicas.
- Failure Detector: the algorithms used were LFA, LFAe, ACT and CTA with δ equals to 1 ms, 4 ms, 16 ms, 50 ms, 100 ms, 200 ms and 300 ms, depending on the experiment performed (see section 5.2.4). As mentioned above, the FD uses a dedicated physical network.
- Opponent: when assessing FD speed metrics (detection and stabilization time), the *opponent* was configured to introduce processes failures (crash-stop and crash-recovery models) via a mechanism that emulates a total failure in the process communication, by dropping all the packets sent/received by the failing process. When assessing FD accuracy metrics (average mistake rate and good period duration) the *opponent* was configured to emulate distinct networks conditions on a failure-free environment. It emulates distinct conditions by dropping and delaying packets based on two distributions: *EnumeratedIntegerDistribution* [5] and *NormalDistribution* [7]. All details on how the *opponent* was configured for each experiment can be seen in section 5.2.

5.1.3 Data Management

The appropriate *data management*, as recognized by FAPESP [9], facilitates the reproducibility of the results and allows the promotion of new research. Besides that, it helps to carry out new analyzes, with the execution of other tests or methods of analysis. Since the beginning of this work, we focused on following *data management* good practices by structuring the research in a way that:

1. All relevant files were in a version control system, Git [10], in a safe web-based repository, Bitbucket [4], accessible to any interested part (Figure 5.1).

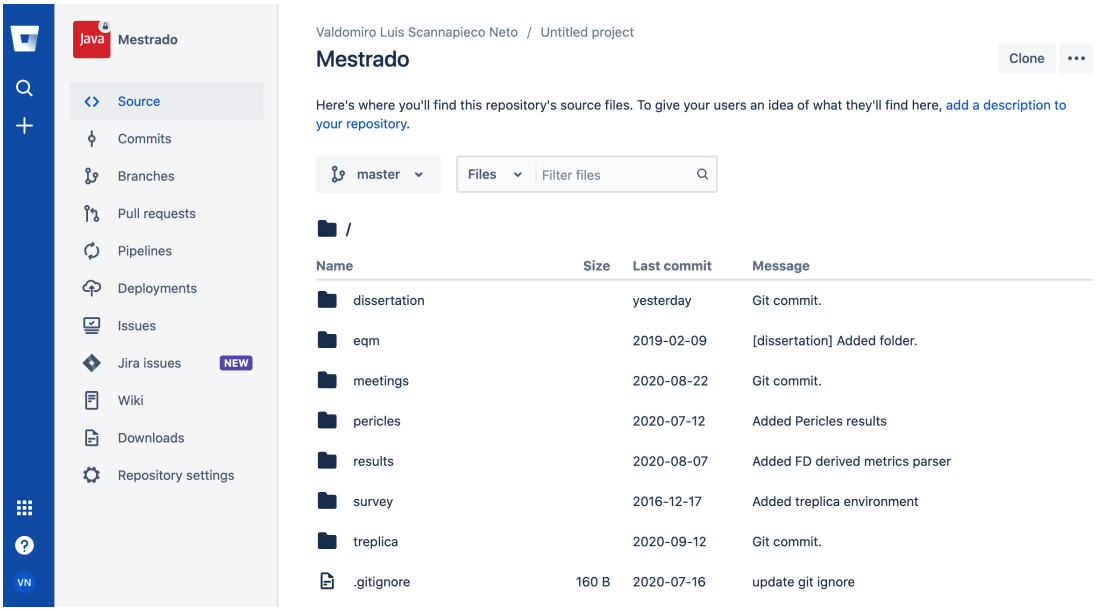

Figure 5.1: Source code in Bitbucket/Git.

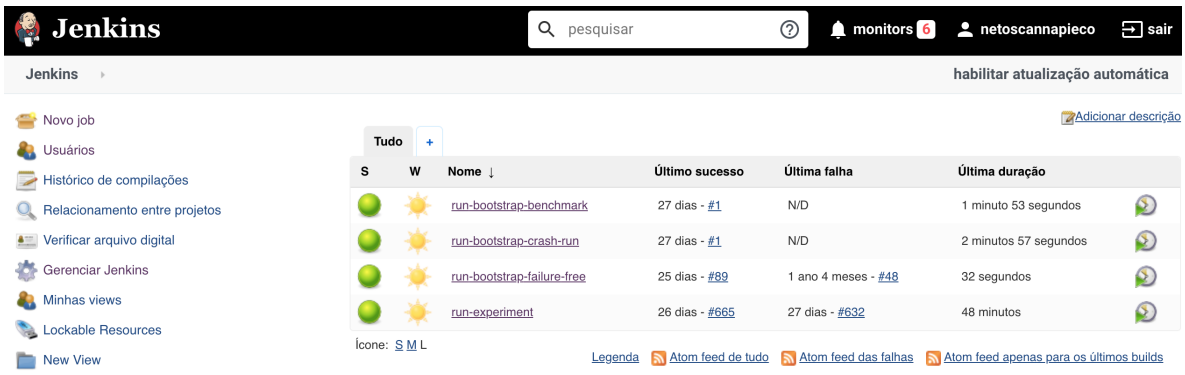

Figure 5.2: Experiments automation via Jenkins and Ansible.

2. All experiments, although we were dealing with a cluster of several machines and a dozen of tunable settings, were simple to setup, run and analyse by making use of tools such as Jenkins [11] and Ansible [1] (Figure 5.2).

3. All experiments settings and results were stored in both Jenkins and Git making the research transparent to all collaborators.

More specifically regarding *experiments automation*, this approach was essential to the progress of the work since it reduced significantly the complexity of running all experiments variations and measuring their performance accurately.

Experiments automation

Jenkins [11] is the leading open source automation server. It mainly helps to automate the non-human part of the software development process, with continuous integration and facilitating technical aspects of continuous delivery. We used Jenkins in our *experiments automation* to execute the following steps:

1. Setup the experiment settings via Jenkins interface (Figure 5.3).

2. Download the latest source code and config file from Bitbucket/Git.

3. Run unit tests to guarantee the current source code is stable, compile and package the solution in a JAR file (by calling the Apache Maven [12] tool).

4. Make sure that all machines on the cluster are ready to run the experiments by (i) terminating all Java process currently in execution, (ii) cleaning up all Treplica and FDs residual files, (iii) verifying if NTP is up and running and (iv) checking current CPU and memory usage. All the steps above were executed via Ansible [1] before each experiment iteration.

5. Distribute the package and config file among all machines, execute the experiments and gather the results.

6. Run statistical analysis over the aggregated results.

The Ansible [1] automation tool was the chosen one to manage the cluster machines since it's agentless (connecting remotely via SSH) which is a great feature, meaning that the nodes crucial to our experiments weren't overloaded.

Jenkins and Ansible ran in a dedicated machine, other than the nodes used in the experiment. Basically, after setting up the tunable settings, the experiment could be started by clicking on a "play" button and, minutes/hours later (depending on the number of iterations and duration) the results were ready to be analyzed.

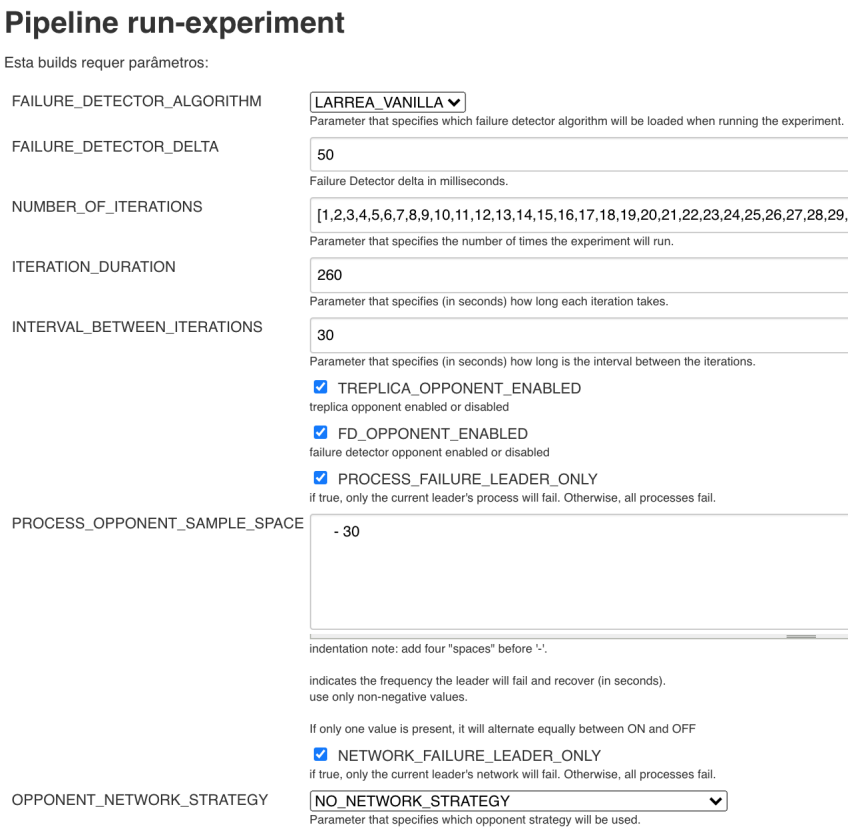

Figure 5.3: Jenkins experiment settings.

5.2 Experiments

In order to proceed with the comparative study of failure detectors, experiments were built to measure (i) the failure detector speed - how fast it detects a failure - (ii) the FD accuracy - how well it avoids mistakes - and (iii) how the different *network/process* environment conditions influence consensus performance.

5.2.1 FD speed assessment

Based on Chen et al. [30] work, two important metrics were defined to compare the FDs speed:

- TD_m (majority detection time): moment after the leader failure in which the majority of correct processes detect that the leader process has failed.
- TE_m (majority stabilization time): moment after the leader failure in which the majority of correct processes stabilize on the new leader.

The metrics were defined considering the majority of processes because Paxos [56] requires a simple majority of correct processes to work and, this way, we focused on metrics more appropriate to the system under test.

Figure 5.4 explains how TD_m and TE_m metrics work. It depicts an LFA execution with three processes *P1*, *P2* and *P3*, whose identifiers are ordered in a global and non-decreasing order with ID(*P1*) < ID(*P2*) < ID(*P3*). In the beginning of the execution, all three processes are correct, with *P1* as the leader (lowest *id*), sending messages periodically to *P2* and *P3*. At time *T0*, process *P1* fails and stops sending heartbeats. After *T0*, the first process to detect the failure is *P3* at time *T1*. According to the LFA algorithm, from this instant on, *P3* starts sending heartbeats to the other processes in an attempt to become the new leader. At time *T2*, process *P2* also detects *P1* failure and, like *P3*, tries to become the new leader by also sending heartbeats. At *T3*, process *P3* receives the first message from *P2* and starts considering *P2* as leader (ID(*P2*) < ID(*P3*)). From *T3* on, the FD algorithm is stable again, with *P2* as the new elected leader. The detection time T_{DP2} and T_{DP3} are, respectively, the amount of time demanded for *P2* and *P3* to detect *P1* failure. TD_m is the amount of time all correct processes took to detect the leader failure, whereas TE_m is the amount of time all correct processes took to stabilize, agreeing on a new leader.

To obtain the FD speed metrics, tests exploring failures on the *leader process* were implemented. *Process* failures were injected via a mechanism of the developed *opponent* that emulates a total failure on the leader process communication. When enabled, the mechanism prevented this process from sending/receiving any network message for a period of time. For practical purposes, network failures, when taking efect for a period of time, are indistinguishable from process or machine failures [45]. Both the **crash-stop** and **crash-recovery** models were adopted in the experiments, in other words, the *opponent* mechanism either disabled the message exchanges permanently or switched between

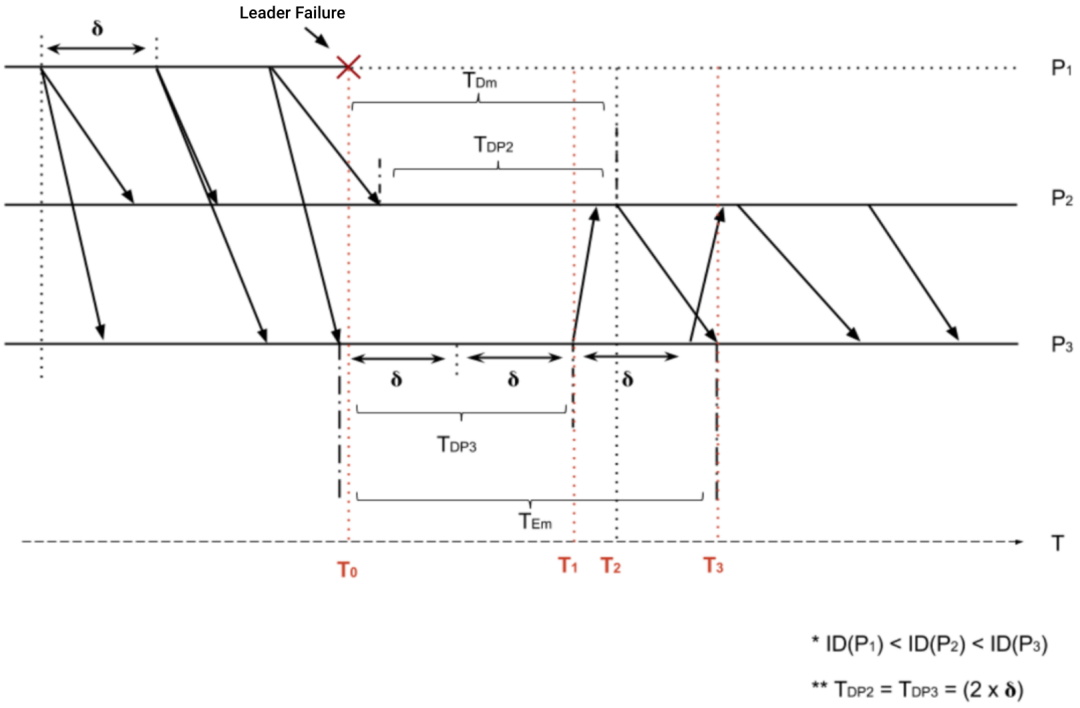

Figure 5.4: Larrea et al. [59] algorithm execution with leader failure.

enabled and disabled periodically. Furthermore, the cluster ***network* is considered perfect**, giving that the *opponent* was only acting on the packets of the leader process.

During the **crash-recovery** experiments the leader switched between periods of crash and periods of recovery. The periods adopted were 5, 10, 15, 20, 25 and 30 seconds. Therefore, for each iteration (600 seconds), several *process* failures/recoveries occurred resulting in several TD_m and TE_m values. On the **stop-crash** experiment the leader crashed at 450 seconds, not recovering anymore. The failure detector δ was equal to 50 ms.

5.2.2 FD accuracy assessment

Two important metrics were used to measure the FDs accuracy as detailed in the QoS section 3.2 and highlighted below:

- λ_M (average mistake rate): measures the rate at which a failure detector make mistakes, i.e., it is the average number of *S-transitions* per time unit. This metric is important to long-lived applications where each failure detector mistake (each *S-transition*) results in a costly interrupt.

- T_G (good period duration): measures the length of a good period. More precisely, T_G is a random variable representing the time that elapses from a *T-transition* to the next *S-transition*. Many applications can make progress only during good periods - periods in which the failure detector makes no mistakes.

These metrics are relevant to the current work when assessing the impact of FD's accuracy on consensus. Consensus-based applications like Treplica, the replication library adopted to assess consensus performance in this work, may rely on good periods to make progress and failure detector mistakes can be costly. It's important to notice that, since we're are dealing with a cluster of several machines, we understand that a *T-transition* happens when a majority of processes trust in the leader and that a *S-transition* occurs when a majority of processes does not trust in the leader.

Based on Chen et al. [30], all accuracy metrics were obtained in **failure-free** runs (***processes* are considered perfect**). In order to achieve that, the *opponent* mechanism that emulates a total failure on the leader process communication, used to obtain the speed metrics, was disabled. Nevertheless, the *opponent* acts here by injecting *network* failures via packets delays and drops following equiprobable and normal distributions.

Equiprobable experiments

Seven equiprobable *network* strategies, making use of the *EnumeratedIntegerDistribution* class, were arbitrarily chosen to test the FDs accuracy under distinct network conditions:

1. **(1)P(1)10DL**: 50% of chance to pass and 50% of chance to have 10 ms of delay.
2. **(1)P(1)40DL**: 50% of chance to pass and 50% of chance to have 40 ms of delay.
3. **(1)P(1)60DL**: 50% of chance to pass and 50% of chance to have 60 ms of delay.
4. **(1)P(1)DR**: 50% of chance to pass and 50% of chance to be dropped.
5. **(4)P(1)DR**: 80% of chance to pass and 20% of chance to be dropped.
6. **(19)P(1)DR**: 95% of chance to pass and 5% of chance to be dropped.
7. **(16)P(1)40DL(1)50DL(1)60DL(1)DR**: 80% of chance to pass, 5% of 40 ms of delay, 5% of 50 ms of delay, 5% of 60 ms of delay and 5% to be dropped.

Distinct FD δs were adopted for the equiprobable experiments: 1 ms, 4 ms, 16 ms and 50 ms.

WAN conditions experiments

Another set of experiments were chosen based on Amir et al. [15] work. [15] shows that the average single link latency between two distant sites - from Mae East and UCSB, a Wide Area Network (WAN) - was of 136 ms with, roughly speaking, 20 ms of standard deviation. Based on it, the following strategies were also added:

1. **(NOR-136-20)DL(0%)DR**: 0% of chance to be dropped and delay following a normal distribution with mean of 136 ms and standard deviation of 20 ms.
2. **(NOR-136-20)DL(5%)DR**: 5% of chance to be dropped and delay following a normal distribution with mean of 136 ms and standard deviation of 20 ms.

3. **(NOR-136-20)DL(10%)DR**: 10% of chance to be dropped and delay following a normal distribution with mean of 136 ms and standard deviation of 20 ms.

4. **(NOR-136-20)DL(20%)DR**: 20% of chance to be dropped and delay following a normal distribution with mean of 136 ms and standard deviation of 20 ms.

Distinct FD δs were also adopted for the WAN conditions experiments: 50 ms, 100 ms, 200 ms and 300 ms.

5.2.3 Consensus performance assessment

In this dissertation, we also assessed how both *process* and *network* failures influenced consensus performance. In order to measure it on the developed testbed using Treplica we analyzed how many entries were added to the distributed HashMap stable storage (load generator) at the end of each iteration. Since each map entry demands a consensus instance, we were able to infer *consensus rate* (number of consensus instances per second). For instance, let's suppose that at the end of an iteration that lasted 600 seconds (40 seconds of warm up plus 560 seconds of load), 1,120 map entries were put into the HashMap. Taking into account this information, plus the fact that we know that the system keeps a linear behavior along the entire experiment execution, we can infer that the *consensus rate* was 2, meaning that 2 consensus instance were achieved per second in this scenario.

Per experiment, since we have 60 iterations, 60 λ_M and *consensus rate* were collected. The number of T_G, and respective durations, varied significantly depending on the experiment settings.

5.2.4 FD δ

As explained in section 3.3, the failure detector δ has to be chosen carefully since a small δ could lead to frequent searches for new trusted processes whereas a big δ could delay a failure detection and, as can be noted, δ is a system parameter that must be regulated according to the average latency of the network.

Given that, we considered (i) the cluster Round-Trip Time (RTT) of less than 1 ms among the machines and (ii) the average packet delay of the opponent on the experiments to define the following values for δ: **1 ms, 4 ms, 16 ms and 50 ms** for the equiprobable experiments and **50 ms, 100 ms, 200 ms and 300 ms** for the WAN conditions experiments.

It's very important to note that in this work we didn't focus on finding the best value for δ in each experiment, however it's expected that system engineers execute this step when setting up the testbed for failure detectors.

5.3 Results

In this section we show the results of our comparative study of four known failure detectors: LFA, LFAe, ACT and CTA. Firstly, we explain how Bootstrapping was adopted to

compare the results with 95% confidence interval. Then we cover, respectively, FD speed, FD accuracy and consensus instances performance results.

5.3.1 Bootstrapping

After carrying out the experiments, a mass of data was obtained that should be analyzed. The main problem found in this part of the work was the lack of knowledge of the statistical distribution of data from the results of the experiments. In addition, even assuming that the data is distributed in a normal distribution (an assumption that in itself can already be completely wrong, leading to false results), the test execution time makes it impossible to collect a large number of data. It is also known that performing a normal statistical analysis with a volume of data that is not sufficiently large can generate wrong conclusions, as demonstrated and exemplified in [29]. The solution found was to use the Bootstraping method [42] to compare the results of the metrics found.

Booststrapping was essential to understand whether the obtained metrics means (TD_m, TE_m, λ_M and consensus rate) among the four FDs assessed were indeed significantly different or happened by chance. Booststrapping basically takes a given sample and either creates new samples by randomly selecting values from the given sample with replacement, or by randomly shuffling labels on the data. Below, two scenarios are exemplified.

Example 1: after an experiment, LFA TD_m mean was 49.05 ms whereas LFAe TD_m mean was 45.39 ms. Thus, the difference of these two means was only 3.66 ms. By running bootstrapping (random sampling with replacement) 100,000 times, the result was the following:

---

61 out of 100000 experiments had a difference of two means greater than or equal to 3.66. The chance of getting a difference of two means greater than or equal to 3.66 is 0061.

Since the probability of getting a difference of two means, greater than or equal to 3.66 ms, was only 0.061%, we can affirm, considering a 95% confidence interval, that the difference between means **was real**. Therefore, in this experiment, LFAe indeed presents a better TD_m (45.39 ms).

Example 2: after another experiment, LFAe TD_m mean was 45.39 ms while ACT TD_m mean was 45.82 ms. Thus, the difference of these two means was only 0.43 ms. By running bootstrapping 100,000 times, the result was the following:

36272 out of 100000 experiments had a difference of two means greater than or equal to 0.43. The chance of getting a difference of two means greater than or equal to 0.43 is 0.36272.

Since the probability of getting a difference of two means, greater than or equal to 0.43 ms, was 36.2%, we can affirm, considering a 95% confidence interval, that the difference **was not real**. Therefore, in this experiment, LFAe and CTA present similar TD_m.

In order to analyse all of the results below, we run bootstrapping 100,000 times and considered a confidence interval of 95%.

5.3.2 FD speed results

Table 5.1 presents FD speed results. Based on the QoS metrics means and bootstrapping results (present on the official repository, see Appendix A) the conclusion was:

- TD_m (majority detection time): on crash-recovery, **LFA**, **LFAe** and **ACT** presented the best and equivalent TD_m performance, with a variation of less than 2 ms among them (very small compared to the δ of 50 ms). On crash-stop, **ACT** had the best detection time with a significant difference when compared to the other FDs. **CTA** showed the worst performance in terms of speed metrics.

- TE_m (majority stabilization time): **ACT** presented the best TE_m on both crash-stop and crash-recovery. **CTA** presented the worst stabilization time.

Therefore, when considering exclusively FD speed results, **ACT** is the best choice. It's one of the FDs with message exchanges complexity of $O(n^2)$ during its whole execution and since each replica maintains an updated local vector with the number of heartbeats received from each process of the system, it's able to detect a leader change (TD_m) and stabilize (TE_m) on the new leader at the same time.

LFA and **LFAe** non-leader processes only start sending heartbeats when the leader failure is detected what increases TE_m. **CTA**, on the other hand, presents a more sophisticated detection mechanism (indexed messages and timing windows) what ends up approximating the detection and stabilization time to 2δ.

			TDm (ms)				**TEm (ms)**			
Leader Process Strategy	**Network Strategy**	**FD δ (ms)**	**LFA**	**LFAe**	**ACT**	**CTA**	**LFA**	**LFAe**	**ACT**	**CTA**
crash 450 sec / stop	Perfect	50	43.08	45.78	36.10	105.48	58.98	62.88	36.10	105.48
crash 5 sec / recover 5 sec	Perfect	50	50.08	48.55	50.29	110.56	68.44	68.89	50.88	110.62
crash 10 sec / recover 10 sec	Perfect	50	54.14	52.05	54.39	114.74	73.43	71.07	55.48	114.75
crash 15 sec / recover 15 sec	Perfect	50	53.17	53.35	54.41	113.15	70.18	71.01	54.54	113.29
crash 20 sec / recover 20 sec	Perfect	50	52.50	50.16	50.44	111.29	69.95	67.45	50.44	111.29
crash 25 sec / recover 25 sec	Perfect	50	51.60	51.22	48.50	111.22	68.05	67.76	48.50	111.22
crash 30 sec / recover 30 sec	Perfect	50	49.47	49.50	47.76	109.94	67.35	66.05	48.83	109.94

Table 5.1: FD speed results (detection and stabilization time).

5.3.3 FD accuracy results

Tables 5.2 and 5.3 present λ_M (average mistake rate) results for the equiprobable and WAN conditions accuracy experiments. Based on this QoS metric means and bootstrapping results (present on the official repository, see Appendix A), we identified that:

- among all FD algorithms, **CTA** consistently presented the best accuracy performance (lowest mistake rates) when the *network* was subject to packets delays and/or drop, whereas **ACT** consistently showed the worst performance in terms of accuracy metrics.

- the more aggressive (lower) the δ, the higher the mistake rate was for **LFA**, **LFAe** and **ACT** in such networks.

			λ_M (per second)			
Leader Process Strategy	**Network Strategy**	**FD δ (ms)**	**LFA**	**LFAe**	**ACT**	**CTA**
Perfect	(1)P(1)10DL	1	29.47	36.48	122.75	0
Perfect	(1)P(1)10DL	4	20.60	19.67	51.98	0
Perfect	(1)P(1)10DL	16	5.92	6.63	10.80	0
Perfect	(1)P(1)10DL	50	0.07	0.11	0.19	0
Perfect	(1)P(1)40DL	1	31.53	29.62	123.52	0
Perfect	(1)P(1)40DL	4	27.46	27.20	52.12	0
Perfect	(1)P(1)40DL	16	6.53	6.20	14.41	0.05
Perfect	(1)P(1)40DL	50	3.15	3.02	4.08	0.12
Perfect	(1)P(1)60DL	1	30.82	32.18	123.09	0
Perfect	(1)P(1)60DL	4	25.72	28.19	52.05	0
Perfect	(1)P(1)60DL	16	6.82	6.98	14.32	0.86
Perfect	(1)P(1)60DL	50	2.64	2.74	5.54	2.41
Perfect	(1)P(1)DR	1	104.39	103.68	44.09	0
Perfect	(1)P(1)DR	4	35.98	33.00	37.32	0
Perfect	(1)P(1)DR	16	8.73	9.09	13.22	3.19
Perfect	(1)P(1)DR	50	3.17	2.96	4.43	3.30
Perfect	(4)P(1)DRL	1	42.45	42.78	125.79	0
Perfect	(4)P(1)DR	4	36.89	34.50	56.41	0
Perfect	(4)P(1)DR	16	10.54	10.16	13.90	0.03
Perfect	(4)P(1)DR	50	3.50	3.44	4.40	0.91
Perfect	(19)P(1)DR	1	3.16	2.19	52.54	0
Perfect	(19)P(1)DR	4	6.50	5.09	12.36	0
Perfect	(19)P(1)DR	16	3.41	3.25	3.05	0
Perfect	(19)P(1)DR	50	1.13	1.17	0.97	0.04
Perfect	(16)P(1)40DL(1)50DL(1)60DL(1)DR	1	26.53	23.35	111.79	0
Perfect	(16)P(1)40DL(1)50DL(1)60DL(1)DR	4	26.40	25.70	46.50	0
Perfect	(16)P(1)40DL(1)50DL(1)60DL(1)DR	16	8.69	8.58	11.80	0
Perfect	(16)P(1)40DL(1)50DL(1)60DL(1)DR	50	3.07	3.02	4.08	0.19

Table 5.2: Average mistake rate for the equiprobable experiments.

			λ_M (per second)			
Leader Process Strategy	**Network Strategy**	**FD δ (ms)**	**LFA**	**LFAe**	**ACT**	**CTA**
Perfect	(NOR-136-20)DL(0%)DR	50	1.50	1.39	4.81	0
Perfect	(NOR-136-20)DL(0%)DR	100	0.24	0.26	0.92	0
Perfect	(NOR-136-20)DL(0%)DR	200	0.09	0.09	0.24	0
Perfect	(NOR-136-20)DL(0%)DR	300	0	0	0	0
Perfect	(NOR-136-20)DL(5%)DR	50	2.15	2.18	6.01	0
Perfect	(NOR-136-20)DL(5%)DR	100	0.66	0.66	1.67	0
Perfect	(NOR-136-20)DL(5%)DR	200	0.32	0.33	0.61	0
Perfect	(NOR-136-20)DL(5%)DR	300	0.16	0.16	0.19	0.01
Perfect	(NOR-136-20)DL(10%)DR	50	2.93	2.91	6.92	0
Perfect	(NOR-136-20)DL(10%)DR	100	1.11	1.12	2.36	0
Perfect	(NOR-136-20)DL(10%)DR	200	0.55	0.54	0.92	0
Perfect	(NOR-136-20)DL(10%)DR	300	0.30	0.30	0.40	0
Perfect	(NOR-136-20)DL(20%)DR	50	4.32	4.30	8.04	0
Perfect	(NOR-136-20)DL(20%)DR	100	1.91	1.94	3.39	0
Perfect	(NOR-136-20)DL(20%)DR	200	0.96	0.95	1.54	0
Perfect	(NOR-136-20)DL(20%)DR	300	0.55	0.55	0.81	0

Table 5.3: Average mistake rate for the WAN conditions experiments.

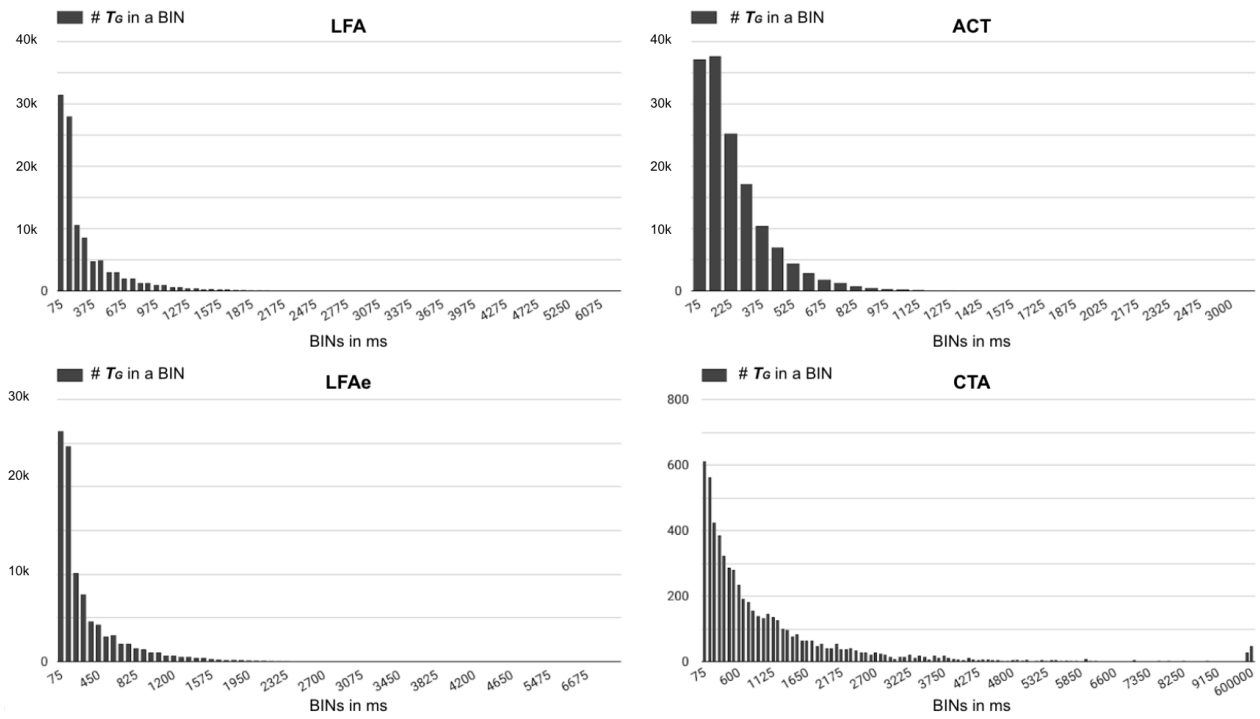

Figure 5.5: T_G histograms for **(1)P(1)40DL** (δ equals to 50 ms).

The same **CTA** good performance can be seen when considering the T_G *(good period duration)* accuracy metric. Figures 5.5, 5.6 and 5.7 show the T_G histograms for (i) **(1)P(1)40DL** (δ equals to 50 ms), (ii) **(4)P(1)DR** (δ equals to 50 ms) and (iii) **(NOR-136-30)DL(20%)DR** (δ equals to 300 ms) respectively. In order to plot these histograms we created BINs of 75 ms, time normally enough to complete a consensus instance in our environment (based on the crash 450 sec / stop experiment, Table 5.4).

By analysing the histograms, it's clear that **LFA**, **LFAe** and **ACT** good periods are more concentrated on shorter durations, whereas **CTA** T_G, is more sparsed or concentrated on longer durations. Lower λ_M and higher T_G, as with **CTA** (the FD with most elaborated detection mechanism, by making use of indexed messages and timing windows), can benefit applications in which leader changes are costly and rely on good periods to progress. That significantly influenced consensus instances performance as shown in section 5.3.4.

LFA and **LFAe** present the two undesirable characteristics mentioned in 3.3.4 that **CTA** attempts to tackle (specially the dependency on past heartbeats), what compromises their performance in case of delays and/or drops. Finally, **ACT** is the most sensitive FD, capable of great speed performance (see 5.3.2) but with poor performance in terms of accuracy.

5.3.4 Consensus performance results

Tables 5.4, 5.5 and 5.6 present consensus performance results. Table 5.4 presents the consensus performance results for crash-stop and crash-recovery whereas tables 5.5 and 5.6

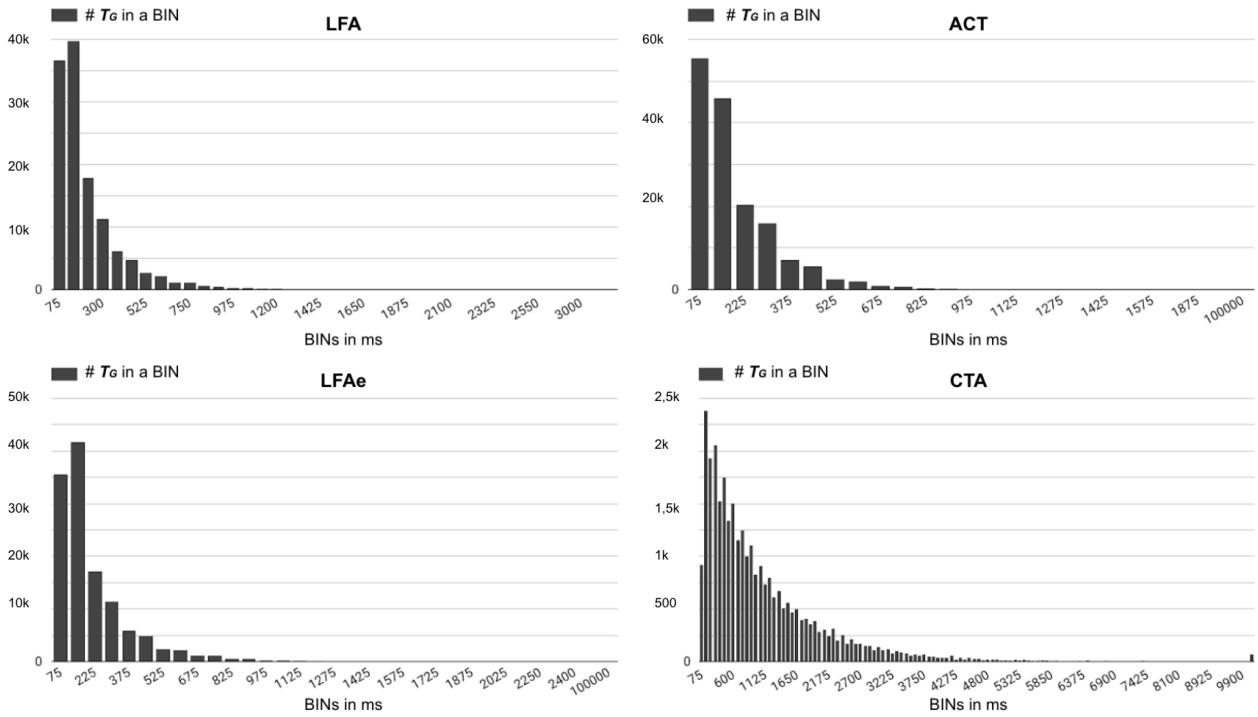

Figure 5.6: T_G histograms for **(4)P(1)DR** (δ equals to 50 ms).

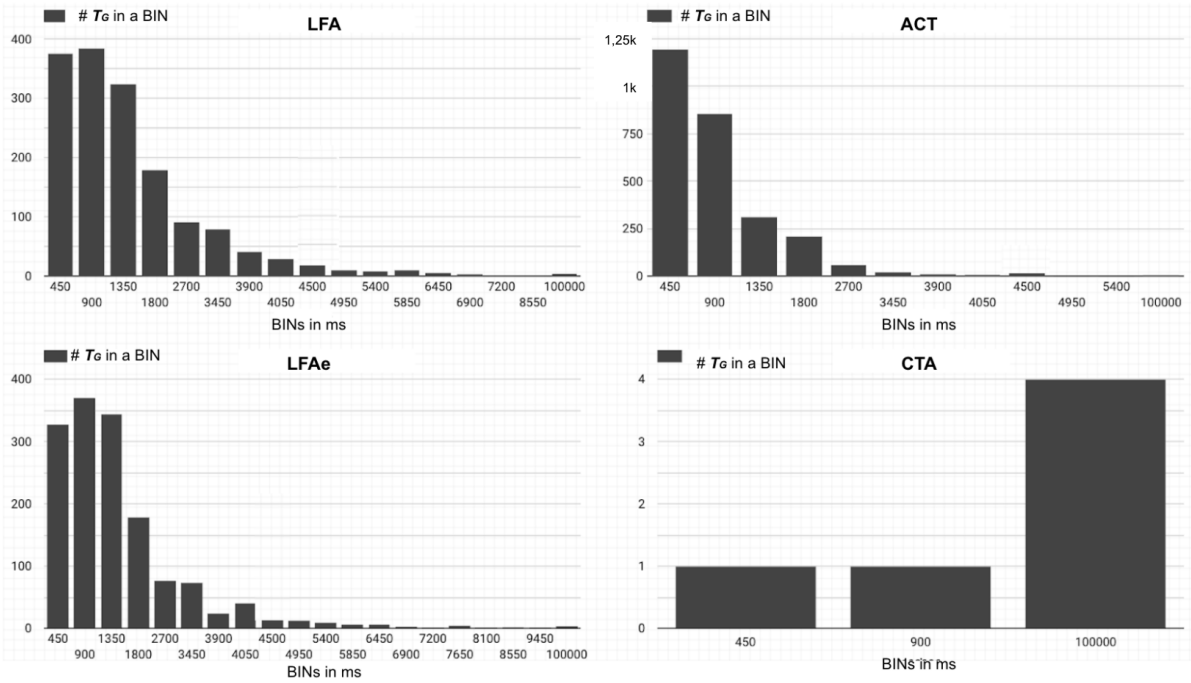

Figure 5.7: T_G histograms for **(NOR-136-30)DL(20%)DR** (δ equals to 300 ms).

present the results for the equiprobable and WAN conditions accuracy experiments.

			Consensus Rate			
Leader Process Strategy	**Network Strategy**	**FD δ (ms)**	**LFA**	**LFAe**	**ACT**	**CTA**
crash 450 sec / stop	Perfect	50	15.12	15.17	15.37	15.29
crash 5 sec / recover 5 sec	Perfect	50	14.38	14.28	14.05	14.15
crash 10 sec / recover 10 sec	Perfect	50	14.75	14.73	14.85	14.78
crash 15 sec / recover 15 sec	Perfect	50	14.98	14.97	14.98	14.96
crash 20 sec / recover 20 sec	Perfect	50	15.03	15.00	15.19	14.83
crash 25 sec / recover 25 sec	Perfect	50	15.06	15.01	15.19	15.06
crash 30 sec / recover 30 sec	Perfect	50	15.20	15.00	15.33	15.33

Table 5.4: *Consensus rate*: crash-stop and crash-recovery strategy.

			Consensus Rate			
Leader Process Strategy	**Network Strategy**	**FD δ (ms)**	**LFA**	**LFAe**	**ACT**	**CTA**
Perfect	(1)P(1)10DL	1	0	0	0	12.53
Perfect	(1)P(1)10DL	4	0	0	0	12.41
Perfect	(1)P(1)10DL	16	0.01	0.01	0.08	12.41
Perfect	(1)P(1)10DL	50	4.92	5.32	9.06	12.36
Perfect	(1)P(1)40DL	1	0	0	0	7.44
Perfect	(1)P(1)40DL	4	0	0	0	7.40
Perfect	(1)P(1)40DL	16	0	0	0.01	6.43
Perfect	(1)P(1)40DL	50	0.17	0.22	0.62	5.77
Perfect	(1)P(1)60DL	1	0	0	0	5.80
Perfect	(1)P(1)60DL	4	0	0	0	5.69
Perfect	(1)P(1)60DL	16	0	0	0.01	3.08
Perfect	(1)P(1)60DL	50	0.03	0.05	0.04	1.41
Perfect	(1)P(1)DR	1	0	0	0	0
Perfect	(1)P(1)DR	4	0	0	0	0
Perfect	(1)P(1)DR	16	0	0	0	0
Perfect	(1)P(1)DR	50	0	0	0	0
Perfect	(4)P(1)DR	1	0	0	0	1.03
Perfect	(4)P(1)DR	4	0	0	0	1.02
Perfect	(4)P(1)DR	16	0	0	0	0.78
Perfect	(4)P(1)DR	50	0.04	0.04	0.01	0.18
Perfect	(19)P(1)DR	1	0	0	0	9.58
Perfect	(19)P(1)DR	4	0	0	0.04	9.56
Perfect	(19)P(1)DR	16	0.85	0.90	1.74	9.52
Perfect	(19)P(1)DR	50	3.98	4.07	4.51	9.05
Perfect	(16)P(1)40DL(1)50DL(1)60DL(1)DR	1	0	0	0	7.01
Perfect	(16)P(1)40DL(1)50DL(1)60DL(1)DR	4	0	0	0	7.01
Perfect	(16)P(1)40DL(1)50DL(1)60DL(1)DR	16	0	0	0.02	6.89
Perfect	(16)P(1)40DL(1)50DL(1)60DL(1)DR	50	0.28	0.28	0.33	5.30

Table 5.5: *Consensus rate*: equiprobable experiments.

Based on *consensus rate* means and bootstrapping results it was possible to observe that, for both **crash-stop** and **crash-recovery** models (*network* considered perfect), there was no relevant difference in *consensus rate* among the different FDs. Moreover, as expected, the more crashes during an experiment the lower the *consensus rate* was, given the time it takes for the system to stabilize on the new leader.

On the other hand, there was a different scenario on the **failure-free** model, in which the *leader process* is perfect but the *opponent* is actively acting on the *network*. It was observed that Treplica consistently presented the best *consensus rate* performance when the underlying FD was **CTA**, except in a few scenarios of the WAN conditions experiments. **CTA** also presented positive *consensus rate* in the majority of experiments - different from **LFA**, **LFAe** and **ACT** which present several zeroed rates - meaning it's a FD algorithm capable of progressing despite the *network* conditions, what is in line with the great accuracy QoS metrics (λ_M and T_G) we observerd in section 5.3.3 .

			Consensus Rate			
Leader Process Strategy	**Network Strategy**	**FD δ (ms)**	**LFA**	**LFAe**	**ACT**	**CTA**
Perfect	(NOR-136-20)DL(0%)DR	50	0	0	0	0.17
Perfect	(NOR-136-20)DL(0%)DR	100	0.04	0.02	0.22	0.19
Perfect	(NOR-136-20)DL(0%)DR	200	0.38	0.37	0.88	0.20
Perfect	(NOR-136-20)DL(0%)DR	300	1.18	1.19	0.70	0.14
Perfect	(NOR-136-20)DL(5%)DR	50	0	0	0	0.17
Perfect	(NOR-136-20)DL(5%)DR	100	0.01	0.01	0.04	0.19
Perfect	(NOR-136-20)DL(5%)DR	200	0.17	0.16	0.45	0.21
Perfect	(NOR-136-20)DL(5%)DR	300	0.83	0.82	0.38	0.12
Perfect	(NOR-136-20)DL(10%)DR	50	0	0	0	0.15
Perfect	(NOR-136-20)DL(10%)DR	100	0	0	0.01	0.17
Perfect	(NOR-136-20)DL(10%)DR	200	0.07	0.07	0.15	0.18
Perfect	(NOR-136-20)DL(10%)DR	300	0.41	0.41	0.15	0.09
Perfect	(NOR-136-20)DL(20%)DR	50	0	0	0	0.09
Perfect	(NOR-136-20)DL(20%)DR	100	0	0	0	0.10
Perfect	(NOR-136-20)DL(20%)DR	200	0.02	0.02	0	0.10
Perfect	(NOR-136-20)DL(20%)DR	300	0.09	0.09	0.01	0.04

Table 5.6: *Consensus rate*: WAN conditions experiments.

Chapter 6

Related Work

The related work mentioned in this section illustrates part of the current effort employed to propose new failure detection algorithms and assess them, along with existing ones, in distinct networks.

Gumerato [49] presents a comparative study of the implementation of four known failure detection algorithms on Local Area Network. The compared FDs were the same of this current dissertation: Larrea et al. [59] vanilla and epoch, Aguilera et al. [14], and Chen et al. [30]. While [49] analyses FD's speed metrics (detection and stabilization time), also using Treplica, on a crash-stop environment, our work firstly proposes an experimental method and testbed to analyse the behavior of FDs under distinct network conditions and processes failures and secondly, with this testbed, assesses (i) FD's speed on both crash-stop and crash-recovery environments, (ii) FD's accuracy metrics (average mistake rate and good period duration) on a failure-free environment and (iii) how these different environment conditions indeed influence consensus performance in terms of consensus rate.

Gumerato's results for FD's speed show no significant difference in the QoS of the studied algorithms. However, our results concluded that ACT - one of the FDs with message exchanges complexity of $O(n^2)$ during its whole execution - is the most recommended one since it presents the best TD_m (along with LFA and LFAe) and the best TE_m. In our understading, due to a misinterpretation of bootstrapping output, the author concluded that the FD's TD_m and TE_m means were not significantly different, however, by analysing Gumerato's experiments data (not included in [49]) we confirmed that there were important differences among the QoS means and these differences were indeed being confirmed by bootstrapping. This current work, therefore, contributes by recommending ACT as the best algorithm in terms of speed.

Chen et al. [30] propose several FD implementations relying on the probabilistic behavior of the network systems. The protocol uses arrival times sampled in the recent past to compute an estimation of the arrival time of the next heartbeat. However, a timeout that is set according to this estimation, plus a constant safety margin, does not match the dynamic network behavior well [20]. Bertier FD [20, 21] provides an optimization of the safety margin for Chen FD. It uses a different estimation function, combining Chen's and Jacobson's estimation of the Round-Trip Time (RTT). Bertier FD is primarily designed to be used over LANs, where messages are seldom lost [50]. The φ

FD [50, 37] proposes an approach where the output is a suspicion level on a continuous scale, instead of providing information of a binary nature (trust or suspect) [33]. These three FDs dynamically predict new timeout values based on observed communication delays to improve the performance of the protocols [75].

Falai and Bondavalli [44] assess and compare the QoS provided by a large family of failure detectors on Wide Area Network. It's known that WANs are more hostile environments than LANs, environment adopted by Gumerato [49], making more difficult to realize an accurate and complete failure detection mechanism. Designing applications on a LAN generally takes advantage of a deeper knowledge of the infrastructure and a higher controllability. WANs are characterized by longer transmission delays and higher loss probability. WAN connections show also bigger variability of both delay and loss probability due to the many hops traversed in today packet switching WAN technology.

Several other works propose new failure detectors for LANs and WANs. Tomsic et al. [69] propose the Two Windows Failure Detector (2W-FD) an algorithm which, according to the authors, provides QoS and is able to react to sudden changes in network conditions. Veron et al. [72] study reputation systems as a mean to detect failures. The reputation mechanism allows node cooperation via the sharing of views about other nodes. For instance, if one node has a good connection to the other nodes, it can share its view to slowly connected nodes and this prevent wrong views about failures. Moraes Rossetto et al. [33] propose a FD which takes into account the relevance of each node, for instance, differentiating low impact and redundant nodes from high impact ones when reporting failures.

Xiong et al. [75], Wang et al. [74], Liu et al. [62] and Sahoo et al. [67] study failure detectors for cloud computing systems. Cloud computing is an increasingly important solution for providing services deployed in dynamically scalable cloud networks. Services in the cloud computing networks may be virtualized with specific servers which host abstracted details. Some of the servers are active and available, while others are busy or heavy loaded and the remaining are offline for various reasons. Xiong et al. [75] propose a self-tuning failure detector which uses a general non-manual analysis method to self-tune its corresponding parameters and satisfy its expected QoS requirements. Wang et al. [74] also present a FD for cloud systems which, in turn, uses a layer-based failure detection mechanism. This layer mechanism divides the FD into three layers - application, operating system and hypervisor - with one specific monitor each. If one layer fails, the corresponding monitor will indicate it. Liu et al. [62], design WD-FD, an accrual FD based on Weibull distribution. By using the Weibull distribution to estimate the distribution of heartbeat interarrival time, the WD-FD can adapt well to changing network conditions and the requirements of any number of concurrently running applications. Finally, Sahoo et al. [67] propose RT-PUSH, a Virtual Machine (VM) fault detector based on timeout and deadline for cloud system running real-time tasks. The RT-PUSH failure detector adopts a distributed approach where fault can be detected by both participating VMs and the hypervisor.

On the Internet side, Turchetti et al. [70], propose a FD for Internet applications that is capable of managing different applications with varying QoS metrics. It adjusts the fresh point interval either by tuning it to maximum requirement or by equating it

with the greatest common division (GCD) of all freshpoints for all the components. The former assumption works best for remote applications while the later one is more suited for local applications.

The failure detection in wireless networks incur greater computational overhead as a non-responding node does not necessarily indicate failure but, rather, that it could have moved out of range. Nevertheless, some FD construction methods have been published for wireless networks. Elhadef and Boukerche [43], Jin et al. [54], Benkaouha et al. [18] and De Vit et al [34] propose FDs for Mobile Ad-hoc Networks (MANETs). Liu et al. [61] propose a FD for Vehicular Ad-hoc Networks (VANETs), networks in which vehicles can suddenly quit or enter the network and the communication links among them may suffer from signal degradation due to obstacles, changes in vehicle densities, etc. The proposed failure detector employs a detection-result sharing mechanism and groups the nodes according to the architecture of VANETs. Liu and Payton [60] and Greve et al. [48] also present interesting results regarding FDs in mobile environments.

Finally, failure detectors are also studied on Software-Defined Networking (SDN) environments as in Turchetti and Procopio Duarte [71] and Yang et al. [77], IoT environment and Wireless Sensor Network (WSN) as in Yang et al. [76] and Benkaouha et al. [19], respectively, Ambient Assisted Living as in Junior et al. [55] , High-Performance Computing (HPC) platforms as in Bosilca et al. [23] and Zhong et al. [78] and unknown dynamic networks as in Jeanneau et at. [53] .

Assessment methodologies

The above-mentioned work adopt different assessment methodologies to evaluate FDs in diverse networks. Some work [33, 69, 72, 75] use real trace files to assess their FDs in a WAN system. Others [49, 69, 71, 72, 74, 75, 55, 70, 23, 78] use an experimental methodology, setting up machines and their respective networks, to evaluate FDs in LANs, Ambient Assisted Living, Internet Applications and HPC. [47, 61, 19, 76] adopt network simulators such as OMNeT++, NS2 and NS3 and, finally, other studies employ an analytical methodology to perform this assessment.

Chapter 7

Conclusion

This work presents three contributions. Firstly, given that the literature offers a large set of failure detector algorithms but there was not a benchmark or testbed to assess them uniformly, an experimental method is proposed to assess the behavior of FDs under distinct network conditions and processes failures. Secondly, based on the method proposed, a testbed for failure detectors is implemented. The testbed is designed to run on a commodity computer cluster and provides researchers with a complete framework (process/network opponent, failure detector, active replication library and load generator) to assess FDs performance. Thirdly, it performs a comparative study of four known failure detection algorithms: Larrea et al. vanilla and epoch, Aguilera et al., and Chen et al.. FDs speed (detection and stabilization time), FDs accuracy (average mistake rate and good period duration) and consensus rate were the metrics evaluated.

Our results show the following: in terms of speed, we concluded that **ACT** - one of the FDs with message exchanges complexity of $O(n^2)$ during its whole execution - is the most recommended one since, in general, it presents the best TD_m and TE_m. This result is different from Gumerato's one which concludes that there was no significant difference in the speed QoS of the studied algorithms. We understand that there was a misinterpretation of bootstrapping output in Gumerato (see chapter 6) and that ACT is in fact the most recommended algorithm to detect actual processes failures. In terms of QoS accuracy, **CTA** - the FD with the most elaborated detection mechanism, by making use of indexed messages and timing windows - consistently presents the best λ_M and T_G under distinct network conditions. This CTA behavior was significantly positive for consensus performance (measured in terms of *consensus rate*), evidencing the impact of FD QoS at the application level.

Based on the results above, if researchers have an environment subject to packets delays and drops and few processes failures, the **CTA** failure detector should be selected. On the other hand, if researchers have an environment in which the network is stable and processes failures can happen, the **ACT** FD would be the most recommended one.

Future Work

An interesting work that could be performed futurely is (i) the assessment of the failure detectors studied under other distributions representing networks such as cloud comput-

ing systems and wireless systems and (ii) the adoption of adaptive failure detectors as described by Chen et al., since in this current work the FD δ was static.

Bibliography

[1] Ansible. https://www.ansible.com/. Accessed: 2021-01-07.

[2] Apache commons mathematics library. http://commons.apache.org/proper/commons-math/index.html. Accessed: 2021-01-07.

[3] Apache zookeeper. https://zookeeper.apache.org/. Accessed: 2021-01-07.

[4] bitbucket. https://bitbucket.org/. Accessed: 2021-01-07.

[5] Class enumeratedintegerdistribution. http://commons.apache.org/proper/commons-math/javadocs/api-3.6/org/apache/commons/math3/distribution/ EnumeratedIntegerDistribution.html. Accessed: 2021-01-07.

[6] Class exponentialdistribution. https://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math4/distribution/ ExponentialDistribution.html. Accessed: 2021-01-07.

[7] Class normaldistribution. http://commons.apache.org/proper/commons-math/javadocs/api-3.6/org/apache/commons/math3/distribution/ NormalDistribution.html. Accessed: 2021-01-07.

[8] Class poissondistribution. https://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math4/distribution/ PoissonDistribution.html. Accessed: 2021-01-07.

[9] Fapesp. https://fapesp.br/gestaodedados. Accessed: 2021-01-07.

[10] git. https://git-scm.com/. Accessed: 2021-01-07.

[11] Jenkins. https://jenkins.io/. Accessed: 2021-01-07.

[12] Maven. https://maven.apache.org/. Accessed: 2021-01-07.

[13] Probability distributions. https://commons.apache.org/proper/commons-math/userguide/distribution.html. Accessed: 2021-01-07.

[14] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Distributed Algorithms*, pages 126–140. Springer, 1997.

[15] Yair Amir, Claudiu Danilov, and Jonathan Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 327–336. IEEE, 2000.

[16] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rudiger Reischuk. Achievable cases in an asynchronous environment. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 337–346. IEEE, 1987.

[17] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[18] Haroun Benkaouha, Abdelkrim Abdelli, Jalel Ben-Othman, and Lynda Mokdad. Towards an efficient failure detection in manets. *Wireless Communications and Mobile Computing*, 16(17):2939–2955, 2016.

[19] Haroun Benkaouha, Abdelkrim Abdelli, Mohamed Guerroumi, Jalel Ben-Othman, and Lynda Mokdad. Eafd, a failure detector for clustered wsn. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2016.

[20] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 354–363. IEEE, 2002.

[21] Marin Bertier, Olivier Marin, and Pierre Sens. Performance analysis of a hierarchical failure detector. In *null*, page 635. IEEE, 2003.

[22] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 263–275. ACM, 1988.

[23] George Bosilca, Aurelien Bouteiller, Amina Guermouche, Thomas Herault, Yves Robert, Pierre Sens, and Jack Dongarra. Failure detection and propagation in hpc systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 312–322. IEEE, 2016.

[24] Michael F Bridgland and Ronald J Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 52–63. ACM, 1987.

[25] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[26] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, second edition, 2011.

[27] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[28] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[29] Tianshi Chen, Qi Guo, Olivier Temam, Yue Wu, Yungang Bao, Zhiwei Xu, and Yunji Chen. Statistical performance comparisons of computers. *IEEE Transactions on Computers*, 64(5):1442–1455, 2014.

[30] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *Computers, IEEE Transactions on*, 51(5):561–580, 2002.

[31] Benny Chor and Cynthia Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, 5:443–497, 1989.

[32] George F Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed systems: concepts and design.* Pearson, fifth edition, 2012.

[33] Anubis Graciela de Moraes Rossetto, Carlos O Rolim, Valderi Leithardt, Guilherme A Borges, Cláudio FR Geyer, Luciana Arantes, and Pierre Sens. A new unreliable failure detector for self-healing in ubiquitous environments. In *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on*, pages 316–323. IEEE, 2015.

[34] Antônio Rodrigo D De Vit, César Marcon, and Raul Ceretta Nunes. Signal strength as support to mobility detection on failure detectors. In *Proceedings of the Symposium on Applied Computing*, pages 647–650. ACM, 2017.

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

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

[37] Xavier Défago, Péter Urbán, Naohiro Hayashibara, and Takuya Katayama. Definition and specification of accrual failure detectors. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 206–215. IEEE, 2005.

[38] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987.

[39] Danny Dolev, Nancy A Lynch, Shlomit S Pinter, Eugene W Stark, and William E Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM (JACM)*, 33(3):499–516, 1986.

[40] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[41] Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne-Anne Pignolet-Oswald. Never say never probabilistic & temporal failure detectors (extended). IPDPS, 2016.

[42] Bradley Efron. Bootstrap methods: another look at the jackknife. In *Bootstrap methods: another look at the jackknife*, pages 1–26. The annals of Statistics, 1979.

[43] Mourad Elhadef and Azzedine Boukerche. A failure detection service for large-scale dependable wireless ad-hoc and sensor networks. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pages 182–189. IEEE, 2007.

[44] Lorenzo Falai and Andrea Bondavalli. Experimental evaluation of the qos of failure detectors on wide area network. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 624–633. IEEE, 2005.

[45] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[46] Hector Garcia-Molina. Elections in a distributed computing system. *Computers, IEEE Transactions on*, 100(1):48–59, 1982.

[47] Carlos Gomez-Calzado, Mikel Larrea, Iratxe Soraluze, Alberto Lafuente, and Roberto Cortinas. An evaluation of efficient leader election algorithms for crash-recovery systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 180–188. IEEE, 2013.

[48] Fabíola Greve, Pierre Sens, Luciana Arantes, and Véronique Simon. A failure detector for wireless networks with unknown membership. In *European Conference on Parallel Processing*, pages 27–38. Springer, 2011.

[49] Péricles Pompermayer Gumerato. Detetores de falhas em aglomerados: um estudo comparativo. *Dissertação de Mestrado*, 2015.

[50] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The φ accrual failure detector. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 66–78. IEEE, 2004.

[51] Felix Hupfeld, Björn Kolbeck, Jan Stender, Mikael Högqvist, Toni Cortes, Jonathan Martí, and Jesús Malo. Fatlease: scalable fault-tolerant lease negotiation with paxos. *Cluster Computing*, 12(2):175–188, 2009.

[52] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[53] Denis Jeanneau, Thibault Rieutord, Luciana Arantes, and Pierre Sens. Solving k-set agreement using failure detectors in unknown dynamic networks. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1484–1499, 2016.

[54] Ruofan Jin, Bing Wang, Wei Wei, Xiaolan Zhang, Xian Chen, Yaakov Bar-Shalom, and Peter Willett. Detecting node failures in mobile wireless networks: a probabilistic approach. *IEEE Transactions on Mobile Computing*, 15(7):1647–1660, 2015.

[55] Airton Jesus Junior, Tarcísio da Rocha, and Edward David Moreno. A failure detector for ambient assisted living. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–4. IEEE, 2018.

[56] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[57] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[58] Butler W Lampson. How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms*, pages 1–17. Springer, 1996.

[59] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on*, pages 52–59. IEEE, 2000.

[60] Dingxiang Liu and Jamie Payton. Adaptive fault detection approaches for dynamic mobile networks. In *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*, pages 735–739. IEEE, 2011.

[61] Jiaxi Liu, Fei Ding, and Dengyin Zhang. A hierarchical failure detector based on architecture in vanets. *IEEE Access*, 7:152813–152820, 2019.

[62] Jiaxi Liu, Zhibo Wu, Jin Wu, Jian Dong, Yao Zhao, and Dongxin Wen. A weibull distribution accrual failure detector for cloud computing. *PloS one*, 12(3):e0173666, 2017.

[63] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, volume 4, pages 8–8, 2004.

[64] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[65] Marcia Pasin, Stéphane Fontaine, and Sara Bouchenak. Failure detection in large scale systems: A survey. In *Network Operations and Management Symposium Workshops, 2008. NOMS Workshops 2008. IEEE*, pages 165–168. IEEE, 2008.

[66] Michel Reynal. A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News*, 36(1):53–70, 2005.

[67] Sampa Sahoo, Bibhudatta Sahoo, and Ashok Kumar Turuk. Rt-push: a vm fault detector for deadline-based tasks in cloud. In *Proceedings of the 3rd International Conference on Communication and Information Processing*, pages 196–201. ACM, 2017.

[68] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[69] Alejandro Z Tomsic, Pierre Sens, Joao Garcia, Luciana Arantes, and Julien Sopena. 2w-fd: A failure detector algorithm with qos. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 885–893. IEEE, 2015.

[70] Rogério C Turchetti, Elias P Duarte, Luciana Arantes, and Pierre Sens. A qos-configurable failure detection service for internet applications. *Journal of Internet Services and Applications*, 7(1):9, 2016.

[71] Rogerio C Turchetti and Elias Procopio Duarte. Implementation of failure detector based on network function virtualization. In *Dependable Systems and Networks Workshops (DSN-W), 2015 IEEE International Conference on*, pages 19–25. IEEE, 2015.

[72] Maxime Véron, Olivier Marin, Sébastien Monnet, and Pierre Sens. Repfd-using reputation systems to detect failures in large dynamic networks. In *Parallel Processing (ICPP), 2015 44th International Conference on*, pages 91–100. IEEE, 2015.

[73] Gustavo MD Vieira and Luiz E Buzato. Treplica: ubiquitous replication. In *SBRC: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*. Citeseer, 2008.

[74] Fengwei Wang, Hai Jin, Deqing Zou, and Weizhong Qiang. Fdkeeper: A quick and open failure detector for cloud computing system. In *Proceedings of the 2014 International C* Conference on Computer Science & Software Engineering*, page 14. ACM, 2014.

[75] Naixue Xiong, Athanasios V Vasilakos, Jie Wu, Y Richard Yang, Andy Rindos, Yuezhi Zhou, Wen-Zhan Song, and Yi Pan. A self-tuning failure detection scheme for cloud computing service. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 668–679. IEEE, 2012.

[76] Rui Yang, Shichu Zhu, Yifei Li, and Indranil Gupta. Medley: A novel distributed failure detector for iot networks. In *Proceedings of the 20th International Middleware Conference*, pages 319–331, 2019.

[77] Tsai-Wei Yang and Kuochen Wang. Failure detection service with low mistake rates for sdn controllers. In *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 1–6. IEEE, 2016.

[78] Dong Zhong, Aurelien Bouteiller, Xi Luo, and George Bosilca. Runtime level failure detection and propagation in hpc systems. In *Proceedings of the 26th European MPI Users' Group Meeting*, pages 1–11, 2019.

Appendix A

Testbed Manual

The official repository resulted from this work containing both the testbed for failure detectors and the experiment results is the following **https://bitbucket.org/scannapieconeto/testbedfailuredetectors** as illustrated in Figure A.1.

The technical details on how to (i) generate the testbed JAR file, (ii) edit the testbed config file, (iii) setup the machines that'll be used and (iv) run the experiments using the testbed is also present in the reposity README.md file and shown in Figures A.2, A.3 and A.4 respectively.

The experiment results were also added to the repository as illustrated by Figure A.5.

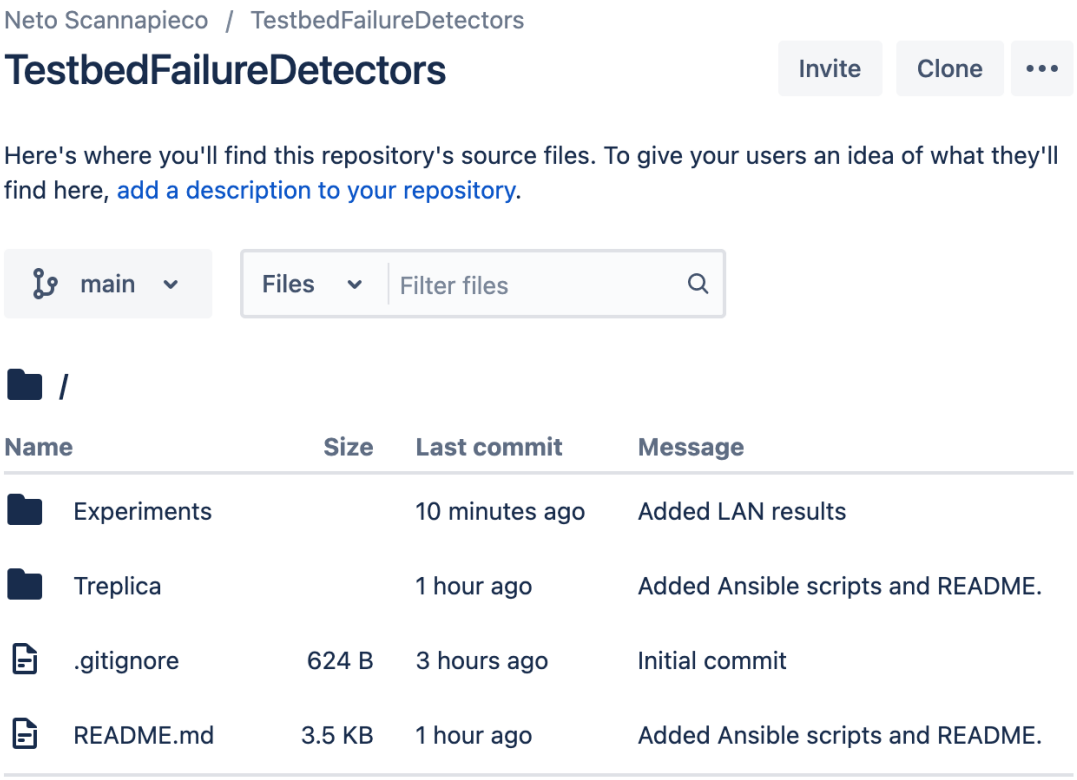

Figure A.1: Official testbed for failure detectors repository.

Figure A.2: How to generate the testbed JAR file.

Edit the Testbed Failure Detector config file

Edit file **testbedfailuredetectors/Treplica/jar-config.yml** with a special attention to the following parameters:

- **algorithm: "${FAILURE_DETECTOR_ALGORITHM}"** failure detector possible algorithms: LARREA_VANILLA, LARREA_EPOCH, AGUILERA, CHEN.
- **delta: ${FAILURE_DETECTOR_DELTA}** failure detector delta (in milliseconds).
- **treplicaOpponentEnabled: ${TREPLICA_OPPONENT_ENABLED}** indicates whether the treplica opponent is enabled or disabled.
- **fdOpponentEnabled: ${FD_OPPONENT_ENABLED}** indicates whether the failure detector opponent is enabled or disabled.
- **processFailureLeaderOnly: ${PROCESS_FAILURE_LEADER_ONLY}** if true, only the current leader's process will fail. Otherwise, all processes fail.
- **processStrategySampleSpace: ${PROCESS_OPPONENT_SAMPLE_SPACE}** indicates the frequency the leader will fail and recover (in seconds). Use only non-negative values.
- **networkFailureLeaderOnly: ${NETWORK_FAILURE_LEADER_ONLY}** if true, only the current leader's network will fail. Otherwise, all processes fail.
- **networkStrategy: "${OPPONENT_NETWORK_STRATEGY}"** network strategy possible values: NO_NETWORK_STRATEGY, ENUMERATED_DISTRIBUTION_NETWORK_STRATEGY, NORMAL_DISTRIBUTION_NETWORK_STRATEGY

Figure A.3: How to edit the testbed config file.

Machines Setup

The Testbed Failure Detector was developed to run on Linux-based machines. Make sure all machines used meet the instructions below:

1. Have NTP (Network Time Protocol) service running.
2. Have IP Multicast configured.
3. Have all directories specified in **testbedfailuredetectors/Treplica/jar-config.yml** created.

Run the Testbed Failure Detector

In order to run the Testbed Failure Detector execute the following steps:

1. Copy both generated **testbedfailuredetectors/Treplica/target/treplica-jar-with-dependencies.jar** and edited **testbedfailuredetectors/Treplica/jar-config.yml** to a directory on the Linux machine.
2. Run command **java -cp treplica-jar-with-dependencies.jar br.unicamp.ic.treplica.clustertools.ReplicatedMapLG** on each machine of the cluster.

Further details of all the commands executed before and after each experiment, including the ones used to extract the Quality of Service metrics, can be seen in directory **testbedfailuredetectors/Treplica/ansible**, specially files **run_experiment.yml** and **run_experiment_iteration.yml**

Figure A.4: Machines setup and commands to run the testbed.

testbedfailuredetectors / Experiments / **WAN**

Name	Size
..	
Normal_Mean_136ms_Standard_Deviation_10_ms_0_loss_100ms_delta.res...	11.21 KB
Normal_Mean_136ms_Standard_Deviation_10_ms_0_loss_200ms_delta.res...	11.22 KB
Normal_Mean_136ms_Standard_Deviation_10_ms_0_loss_300ms_delta.res...	11.23 KB
Normal_Mean_136ms_Standard_Deviation_10_ms_0_loss_50ms_delta.result	11.21 KB
Normal_Mean_136ms_Standard_Deviation_10_ms_10_loss_100ms_delta.re...	11.22 KB

Figure A.5: Experiment results in the repository.