Rodrigo Rocco Barbieri

Achieving non-malicious arbitrary fault tolerance in Paxos through hardening techniques

Sorocaba, SP

4 de Agosto de 2016

Rodrigo Rocco Barbieri

Achieving non-malicious arbitrary fault tolerance in Paxos through hardening techniques

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCCS) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Redes de Computadores e Engenharia de Software.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCCS

Orientador: Gustavo Maciel Dias Vieira

Sorocaba, SP

4 de Agosto de 2016

Rocco Barbieri, Rodrigo

Achieving non-malicious arbitrary fault tolerance in Paxos through hardening techniques / Rodrigo Rocco Barbieri. -- 2016.
63 f. : 30 cm.

Dissertação (mestrado)-Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador: Gustavo Maciel Dias Vieira
Banca examinadora: Luiz Eduardo Buzato, Yeda Regina Venturini
Bibliografia

1. Fault tolerance. 2. Distributed algorithms. 3. Paxos. I. Orientador. II. Universidade Federal de São Carlos. III. Título.

Ficha catalográfica elaborada pelo Programa de Geração Automática da Secretaria Geral de Informática (SIn).
DADOS FORNECIDOS PELO(A) AUTOR(A)

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Rodrigo Rocco Barbieri, realizada em 04/08/2016:

Prof. Dr. Gustavo Maciel Dias Vieira
UFSCar

Prof. Dr. Luiz Eduardo Buzato
UNICAMP

Profa. Dra. Yeda Regina Venturini
UFSCar

Abstract

Due to the widespread adoption of distributed systems when building applications, demand for reliability and availability has increased. These properties can be achieved through replication techniques using algorithms that must be capable of tolerating faults. Certain faults such as arbitrary faults, however, may be more difficult to tolerate, resulting in more complex and resource intensive algorithms that end up being not very practical to use. Using an existing benign fault-tolerant middleware based on Paxos, we propose and experiment with the usage of consistency validation techniques and a distributed validation mechanism to harden it, thus allowing any application built on top of this hardened middleware to tolerate non-malicious arbitrary faults.

Key-words: Fault tolerance. Paxos. Benign faults. Arbitrary faults. Consistency validation. Distributed validation. Hardening. Non-malicious.

Resumo

Devido a crescente adoção de sistemas distribuídos ao se desenvolver aplicações, a demanda por confiabilidade e disponibilidade tem aumentado. Essas propriedades podem ser alcançadas através de técnicas de replicação que utilizam algoritmos capazes de tolerar falhas. Alguns tipos de falhas como falhas arbitrárias, porém, podem ser mais difíceis de tolerar, resultando em algoritmos mais complexos e custosos que acabam não sendo tão viáveis de serem usados. Utilizando um *middleware* tolerante a falhas benignas já existente baseado em Paxos, nós propomos e experimentamos o uso de técnicas de validação de consistência e um mecanismo de validação distribuída para fortalecê-lo, permitindo então que qualquer aplicação desenvolvida em cima deste *middleware* fortalecido tolere falhas arbitrárias não-maliciosas.

Palavras-chave: Tolerância a falhas. Paxos. Falhas benignas. Falhas arbitrárias. Validação de consistência. Validação distribuída. Fortalecimento. Não-malicioso.

List of Figures

List of Tables

Contents

Introduction

Distributed systems have often been used as a basis for a wide range of services and applications. The adoption of this distributed computing model is motivated by the need for satisfying requirements that become indispensable as we become dependent on automated systems, requirements such as reliability and availability. It is quite difficult to guarantee these two requirements in distributed systems due to the possibility of failure of the involved components. One of the most popular approaches to improve a distributed system's availability is replicating the application (GUERRAOUI; SCHIPER, 1996), so failure occurrences do not compromise availability because there are several replicas providing the same application. Reliability is more difficult to guarantee, because the replicated application must not have its state or part of it corrupted when there is a partial failure.

There are several factors that make it more difficult to synchronize the application state between replicas. The easiest ones to tolerate usually are communication problems, crashes or power outages, that prevent a replica from being updated. However, there are other factors that may be much harder to tolerate, such as data corruption, that may cause the replica to display erroneous behavior and incorrect results. It is useful to classify those faults into two main classes: benign faults and arbitrary faults (CACHIN; GUERRAOUI; RODRIGUES, 2011). The first class represents faults that are related to the use of software and hardware components that may stop working at any given time, but which do not deviate from expected behavior. As for the second class, it represents any fault where the components may display any type of behavior, including behavior caused by external malicious attackers. Table 1 lists some examples of faults and failures involved in each fault class (CORREIA et al., 2012).

One of the most well known techniques to implement replication in a consistent way is through active replication. In this model the application can report to the client application that the operation has succeeded only when a minimum number of replicas

Fault class	Faults	Failures
Benign	System crash	Replica unavailable
	Loss of network connectivity	
	Application freeze	
	Power outage	
Arbitrary	Data corruption	Possibly crash, process or display incorrect information
	Human mistakes	
	System-wide bugs	
	Malicious attackers	

Table 1: Example of faults and failures.

has committed the change (SCHNEIDER, 1990). The minimum number of replicas varies according to the algorithm used and the fault class chosen to be tolerated. Active replication has been the focus of research when developing middleware for distributed applications. There are several ways to tolerate faults, solutions often categorize themselves under a fault model that best describes how faults are tolerated, such as crash-stop and crash-recovery. In the crash-stop model the replica is treated as faulty and is thus removed from the distributed system, while in the crash-recovery model the replica is able to recover and continue to participate in the system. Solutions such as Paxos (LAMPORT, 1998), leases (LAMPSON, 1996) and viewstamp (OKI; LISKOV, 1988) are currently implemented in active replication middleware in the crash-recovery fault model each with its own particular approach (RENESSE; SCHIPER; SCHNEIDER, 2014).

Paxos (LAMPORT, 1998) supports benign faults in the crash-recovery model. In order to accomplish this, Paxos uses persistent memory to save its state and uses it to recover in the event of a crash. Paxos guarantees that subsequent updates will force the replica to update its state. Even though Paxos is perfectly tolerant to benign faults, arbitrary faults are able to compromise its reliability and availability requirements. One way of tolerating these faults is adapting the algorithm to a stronger fault model.

A corresponding Paxos algorithm for the arbitrary faults class (LAMPORT, 2011; CASTRO; LISKOV, 2002) is much more costly than its counterpart for the benign faults class, for the following reasons:

1. It performs more message exchanges and disk operations (LAMPORT, 2011);
2. Its usage of encryption increases system's overhead (BHATOTIA et al., 2010);
3. Its implementation is complex (BHATOTIA et al., 2010; BEHRENS; WEIGERT; FETZER, 2013);
4. The root cause of failures cannot be discovered (SCHNEIDER, 1990);
5. It requires that no more than a third of replicas fail (LAMPORT, 2011);
6. It requires the deployment of different platform and applications versions to achieve the heterogeneity required to tolerate system-wide bugs (CORREIA et al., 2012; BHATOTIA et al., 2010);
7. The increase in number of replicas reduces its performance and may cause an increase in fault occurrence (CORREIA et al., 2012; BEHRENS; WEIGERT; FETZER, 2013).

Among the faults tolerated by the algorithm in the arbitrary faults class, there are several types of faults that are not malicious and are relatively common to distributed

applications (CHANDRA; GRIESEMER; REDSTONE, 2007; CORREIA et al., 2012), such as:

Network faults: packet corruption during transmission;

Hardware faults: corrupt read/write operations on main memory or secondary storage;

Programmer faults: mistakes in algorithm implementation;

Operator faults: erroneous behavior due to incorrect configuration.

It is possible to improve a benign fault tolerant algorithm to tolerate the previously mentioned faults, and by choosing to not tolerate malicious faults it is possible to achieve a non-malicious arbitrary fault tolerant algorithm that may be less costly than the arbitrary one, as presented in (BEHRENS; WEIGERT; FETZER, 2013; CORREIA et al., 2012). In this dissertation we show how to harden the benign crash-recovery Paxos through error detection techniques used to tolerate non-malicious arbitrary faults, like redundancy (BEHRENS; WEIGERT; FETZER, 2013), integrity validations (CORREIA et al., 2012) and semantic validations (BHATOTIA et al., 2010). Additionally, we present a new distributed eventual validation mechanism, augmenting Paxos to detect state divergences between replicas, in order to prevent the distributed system from being corrupted by faults occurring in the implementation or operation of the Paxos algorithm itself. Our main contribution is that we have applied these strategies at the middleware level, allowing any application built on top of the middleware to be automatically hardened.

In order to implement, test and validate this research, we used a Paxos-based Java library known as Treplica (VIEIRA; BUZATO, 2008; VIEIRA; BUZATO, 2010) and hardened it by implementing the techniques mentioned above. We used a fault injection library known as AspectJ to inject faults, thus testing and validating the implementation. We were able to harden the originally implemented benign crash-recovery fault model to a crash-stop non-malicious arbitrary fault model by being able to tolerate the previously mentioned arbitrary faults. The end result is that any application built on top of the hardened Treplica is crash-stop non-malicious arbitrary fault tolerant. Our experimental results show that the superior coverage and slightly small performance impact of our implementation justify its feasibility.

The remainder of the dissertation is organized as follows. In Chapter 1 we describe fault models, existing algorithms and related work. In Chapter 2 we describe our proposal, including the framework used and details about the implementation of the techniques. In Chapter 3 we detail the experimental validation and perform an analysis on the observed results.

1 Fault tolerance in distributed systems

Fault occurrences in computer devices are inevitable, thus since before the dawn of distributed systems the research on fault tolerance techniques has been important in contributing to the development and sophistication of current computer systems (GUERRAOUI; SCHIPER, 1996; CACHIN; GUERRAOUI; RODRIGUES, 2011).

Fault tolerance algorithms are often employed in distributed systems to provide an uninterrupted user experience. These algorithms usually include one or several robust error detection and recovering techniques. These long-researched techniques have become commonly used in several algorithms, thus the literature has created the concept of fault models to better classify fault tolerance algorithms.

1.1 Fault models

Fault models abstract the properties a system must satisfy and how a distributed algorithm should tolerate faults. For the benign faults class, there are a few classic fault models (CACHIN; GUERRAOUI; RODRIGUES, 2011) that draw our attention:

crash-stop: a process can only fail by crashing and no longer participates in the algorithm;

crash-recovery: a process fails by crashing but it may later recover to a healthy state and continue to participate in the algorithm.

The fault model for arbitrary faults class (CACHIN; GUERRAOUI; RODRIGUES, 2011) is:

arbitrary: processes can deviate in any way from the algorithm specification.

In the arbitrary fault model, it is impossible for processes to decide whether another process is behaving arbitrarily intentionally or not. We refer to malicious faults when a process is behaving arbitrarily intentionally, through manipulation from a malicious agent, but we have no algorithms or fault models exclusive to tolerate these faults.

The fault models above range from weaker (more strict) to stronger (more general). The stronger the model, the more complex and difficult it is to implement an algorithm. When building a practical distributed system, it is desirable to adopt a fault model that better fits the system and satisfies its requirements for performance and types of faults it must tolerate. However, this is not always the case, since any distributed system that relies on actual computers is prone to arbitrary faults.

1.2 Active replication

In the active replication paradigm (SCHNEIDER, 1990), the application can only report to the client that a request has been successfully processed when a minimum number of replicas has committed the change, as can be seen in Figure 1. This concept enforces a higher level of consistency and availability because if a single replica were to process the request, in the event of a failure the processed request would be lost. Techniques such as state machine modelling allow active replication to be implemented efficiently, allowing the request operation to be replicated as soon as it is received, causing minimal response delay.

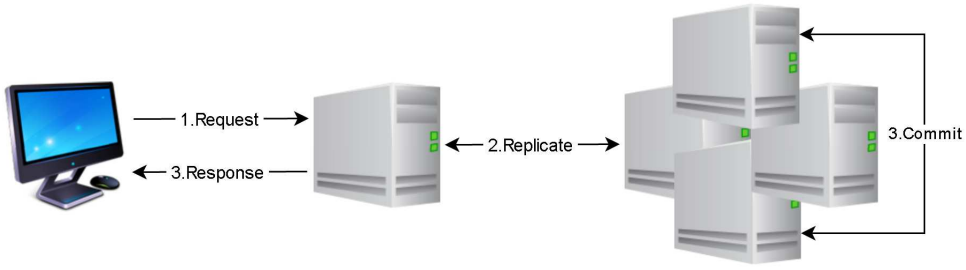

Figure 1: Active replication

Distributed applications built on top of active replication middleware are modeled as deterministic finite state machines. Each system operation is modeled as a state transition, where a state consists of a set of information that includes the previous state and a transition to the current state, as can be seen in Figure 2, where the state machine transitions from state of 1 book to 0 books, upon a button click event. Each replica is a state machine on its own and the algorithm makes use of total order broadcast or consensus algorithm to propagate the transitions in an ordered way. Ultimately, all replicas are kept synchronized in the same state because their transition messages are processed in the same order (SCHNEIDER, 1990).

1.3 Active replication under benign faults

1.3.1 Paxos

Paxos is a consensus-based active replication algorithm proposed for asynchronous systems augmented with failure detectors (LAMPORT, 2001). Paxos also assumes a crash-recovery fault model that tolerates benign faults (LAMPORT, 1998). Replicas agree on a certain value or operation through voting, the decision determines what operation is executed on all replicas. Consensus is reached when the coordinator receives successful votes from the majority of replicas, then the decision is broadcast. This majority is called a quorum.

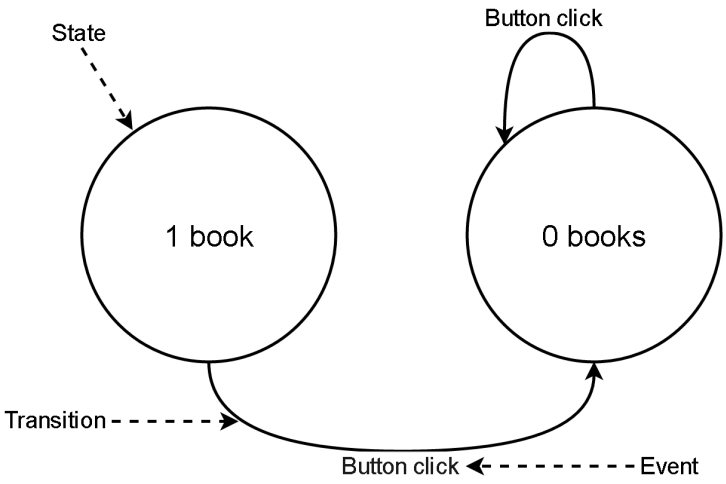

Figure 2: An example state machine

Each new proposal created from a client request is packaged into a uniquely numbered consensus instance, which needs to be decided so the proposal is commited. In order to achieve a decision, a uniquely numbered round for the consensus instance is started, where replicas vote attempting to reach consensus. Several rounds may be necessary until all replicas agree to the proposal.

The Paxos algorithm defines four types of agents each bound to its own responsibilities:

Proposer: responsible for requesting new proposals. The proposer is the entity that listens to client requests, creates a proposal based on the requested operation, and sends to the coordinator to start a new round;

Coordinator: responsible for managing consensus rounds. The coordinator receives proposals, restarts unsuccessful rounds, collects votes and broadcasts the decision;

Acceptor: responsible for voting in active rounds. The acceptor receives round voting requests, votes on them, and reminds the coordinator of previously voted rounds;

Learner: responsible for committing the decided proposal. The learner receives broadcast decisions and commits the proposed operation.

The Paxos algorithm can be described according to the following phases:

1. Pre-Paxos: A replica is elected coordinator through an election algorithm. Whenever a coordinator stops responding, this phase is triggered and a new coordinator is elected.

2. Preparation phase:

a) The proposer creates a proposal from a client request and sends it to the coordinator;

b) The coordinator starts a round numbered higher than any other previous round and announces it to a quorum of acceptors;

c) The quorum of acceptors reply with their highest-numbered round identifier and previously accepted proposal, if any.

3. Accepting phase:

a) The coordinator receives the response from a quorum of acceptors and chooses the round proposal based on the responses;

b) The coordinator sends its proposal or re-sends a previously undecided proposal to a quorum of acceptors;

c) The quorum of acceptors reply with their votes, in case it is their highest-numbered participating round.

4. Learning phase:

a) Once a majority of votes has been received, the coordinator sends the decision to learners;

b) Learners commit the decided proposal upon receiving the decision.

The phases described previously are related to the classic implementation of Paxos, in which there are no concurrent proposals. Multi-Paxos is an implementation that makes use of multiple concurrent proposals, allowing proposals to be chained together and several steps in the algorithm to be skipped for rounds progressing under the same coordinator (LAMPORT, 2001).

In order to satisfy the reliability and availability properties while in the crash-recovery fault model, Paxos must recover its state when a replica fails. Its approach to accomplish this is to save a log of some of its operations in persistent memory, including proposals, votes and decisions, so when the application is restarted, it is able to replay the log and recover the state prior to the crash.

1.3.2 Viewstamp

Viewstamp replication (OKI; LISKOV, 1988) is an algorithm that shares many similarities with Paxos, while the main difference is that it does not use a voting mechanism. It defines a group of replicas which must include a majority, calling it a primary view. One replica in this view is responsible for receiving client requests, creating a transaction and broadcasting it to the other replicas within the view, which must commit the transaction and return success, so the master replica can send the response.

Whenever a replica within the primary view fails to commit a transaction, the view is reconfigured, replacing the faulty replica with another one from the replica pool using a partitioning algorithm. There are three common optimizations employed in this algorithm:

- When a reconfiguration event is triggered, the master replica should remain the same in the new primary view, unless it is the one that failed.
- A read view may be configured, and include a replica from the primary view to ensure consistency. This avoids performance bottleneck on the master replica.
- Data and/or state is periodically transferred to replicas not in the primary view in the background.

Overall this algorithm is simpler and requires less message exchanges than Paxos, achieving slightly better performance (RENESSE; SCHIPER; SCHNEIDER, 2014)

1.3.3 Leases

Locks are common mechanisms used in distributed file systems to guarantee synchronous read and write operations. Locks are not very well suited for systems that may face faults, since the owner of the lock may fail and the resource may become inaccessible. A lease (LAMPSON, 1996) is a lock with an expiration timer. Whenever a replica acquires a lease, it must renew its lease before it expires. If the lease expires, other replicas know that the previous owner has failed.

Leases can also be used to implement consensus. The lease owner writes a certain value in a distributed file system, and all replicas can read that value, while no other replica can overwrite it. Leases are also used in leader election and message ordering algorithms, although some implementations (BURROWS, 2006) prefer to use Paxos to determine the lease owner in the role of master replica in a distributed system.

1.4 Active replication under arbitrary faults

In contrast to benign faults, arbitrary faults create a completely new set of challenges for algorithms to overcome. Previously, algorithms had to handle whether a replica is responding or not. An algorithm that attempts to tolerate arbitrary faults must also handle whether a replica is transmitting a correct message or not. In Figure 3 it is illustrated the effect of arbitrary faults in a consensus algorithm, where leader A sends different proposals to replicas. Replicas B and D do not know if the leader A or replica C is faulty and there is no agreement on whether “1” or “0” is the correct value. In this example, there is no agreement through a majority quorum. The replica broadcasting the incorrect proposal

cannot be ignored because there is no guarantee that the proposal is incorrect and the replica relaying it is the faulty one.

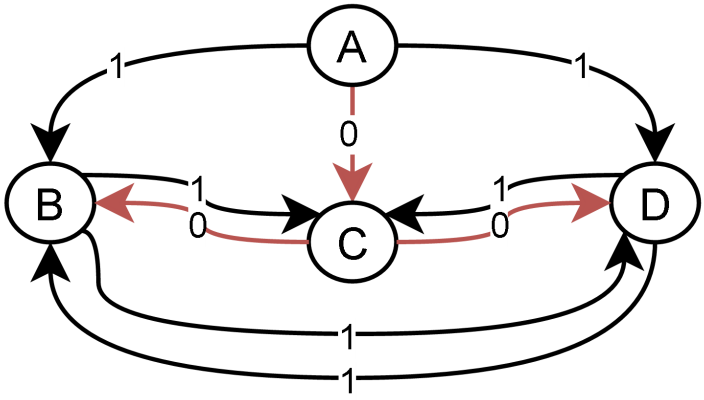

Figure 3: Arbitrary faults in a consensus algorithm

One of the first algorithms to solve consensus in the fail-arbitrary model was described in the paper entitled "The Byzantine Generals Problem" (LAMPORT; SHOSTAK; PEASE, 1982). This paper studies the problem where war generals try to reach an agreement while one of them is a traitor. Due to this work, the arbitrary fault model is often referred to as Byzantine. The main highlight of the problem is how to handle the scenario where a replica may send an incorrect value, either intentionally or not. Some of the distinctions of this algorithm when compared to the traditional benign consensus algorithms are:

1. The quorum size increased from simple majority to more than two thirds of replicas by having overlapping quorums (see Figure 4), for the purpose of isolating faulty replicas. Every replica must be common to two quorums. In the example shown, a seven replicas setup requires a five replica quorum to guarantee that every three quorums intersect;

2. Additional rounds of voting through encrypted broadcast message exchanges, for the purpose of preventing a faulty coordinator from making an incorrect proposal. The operations of encrypting and decrypting messages should prevent malicious attackers and also detect bit flips.

The algorithm proposed by Lamport in (LAMPORT; SHOSTAK; PEASE, 1982) was found to be very complex to implement. The first practical Byzantine algorithm used Paxos in the crash-stop model (SCHNEIDER, 1990), and later a more robust solution was published (CASTRO; LISKOV, 2002), where a heavily modified crash-recovery Paxos is able to tolerate arbitrary faults. This more robust solution made use of Message Authentication Codes (MACs), real-time assumptions, communication protocol through

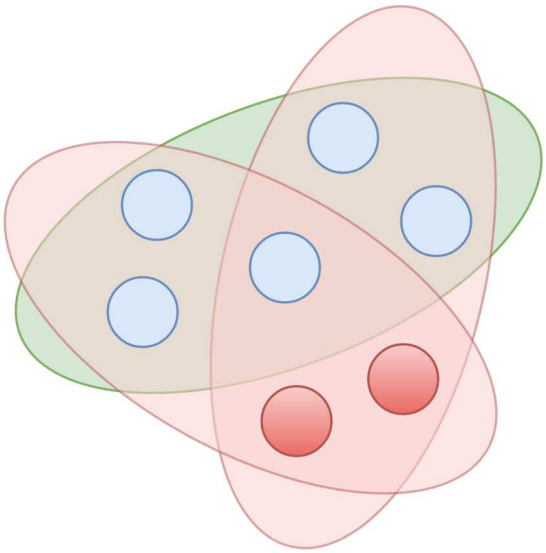

Figure 4: Overlapping quorums

the filesystem, and optimizations necessary to address the recovery of arbitrary faults, such as reconfiguration, garbage collection, checkpointing and state transfer.

Lamport analyzed the algorithm proposed by Castro and Liskov (CASTRO; LISKOV, 2002) and proposed a more general version of the algorithm (LAMPORT, 2011), derived directly from the benign fault tolerant Paxos. The new, more general algorithm, makes use of digital signatures and is changed so that all replicas exchange proposal, vote and decision messages.

Although the crash-recovery Byzantine Paxos algorithm solves consensus in the fail-arbitrary model, its increased performance impact and the fact that malicious faults are being tolerated through solutions orthogonal to distributed systems middlewares (BHATOTIA et al., 2010; CORREIA et al., 2012), contributed to its low-adoption in practical distributed systems. Byzantine Paxos has also been criticized for the following issues:

1. If more than a third of replicas fail, it is not possible to detect that the system has been compromised (SCHNEIDER, 1990). In such scenario, there will be no more overlapping quorums to detect the failing replicas, thus either there will be no more correct decisions, or voting may be manipulated in favor of the faulty replicas;

2. Although faults are tolerated, it may not be possible to find out what triggered it (BHATOTIA et al., 2010; CORREIA et al., 2012). Take for instance the scenario of Figure 3, it is not possible to know if the data got corrupted or a malicious agent is controlling the replica;

3. If there is no heterogeneity in the system, operator and programmer mistakes cannot be detected (CORREIA et al., 2012; BHATOTIA et al., 2010). In such scenario, the

errors will be present across all replicas, so it may end up not being considered a failure at all;

4. Increasing the number of replicas reduces performance and possibly increases fault incidence (CORREIA et al., 2012; BEHRENS; WEIGERT; FETZER, 2013). Take for instance the scenario where the number of replicas in a cluster increases from 7 to 27, the probability of commodity hardware failing is increased, thus there will be more failing voting rounds where consensus will already be taking longer to be reached due to more messages being exchanged between more replicas.

So far there has not been major breakthroughs or new unique algorithms that tolerate arbitrary faults achieving full coverage without extending the algorithm proposed in (LAMPORT; SHOSTAK; PEASE, 1982) or employing hardening techniques (CORREIA et al., 2012; BEHRENS; WEIGERT; FETZER, 2013) on existing benign fault tolerant algorithms while compromising some coverage.

1.5 Non-malicious arbitrary model

Many practical distributed system implementations desire to tolerate arbitrary faults, but would prefer a less performance intensive algorithm than a byzantine one (BHATOTIA et al., 2010; CORREIA et al., 2012; BEHRENS; WEIGERT; FETZER, 2013). While malicious faults are being tolerated using different techniques (BHATOTIA et al., 2010; CORREIA et al., 2012), and based on the premise that any fault model can be hardened to tolerate some arbitrary faults, it is possible to harden the crash-recovery benign model to tolerate non-malicious arbitrary faults, thus achieving the following fault model, as shown in Figure 5:

non-malicious arbitrary: similar to the arbitrary fault model, but malicious faults are not tolerated by the algorithm.

An algorithm for the non-malicious arbitrary fault model can be considered to be less complex than an arbitrary one for not tolerating malicious faults in its implementation. However, the implementation required to tolerate all non-malicious arbitrary faults adds its own complexity to the algorithm. This fault model respects properties similar to the work presented in (BEHRENS; WEIGERT; FETZER, 2013), as follows:

No impersonation: the environment never creates valid messages, except for duplicates. This property assumes that only processes themselves are able to create valid messages, so malicious agents cannot interact with existing processes in a system.

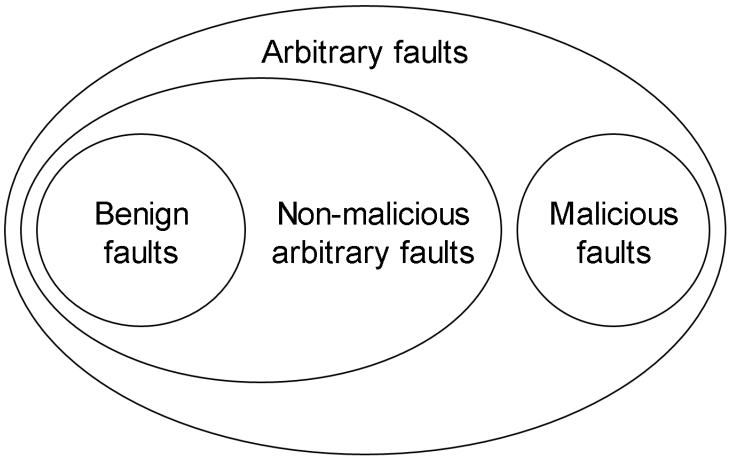

Figure 5: Relationship among fault classes

No propagation: a process that is considered faulty, by either itself or by another process, cannot ever create a valid message. This property assumes that when the process has become faulty, it is not allowed to send any more messages, nor any correct process is allowed to accept messages from a faulty process.

An algorithm for this fault model is expected to tolerate faults caused by data corruption, such as from persistent memory, main memory or network, bugs in the code or configuration mistakes, in addition to benign faults.

In order to satisfy these properties, several techniques are found to be employed on top of a benign fault model:

Enhanced security: firewalls, authentication protocols and network segregation are a few mechanisms used to prevent a malicious agent from accessing system resources, thus satisfying the first property;

Integrity, semantic and distributed validations: redundancy, arithmetic codes, serialization are some strategies used to detect whether non-malicious faults have occurred, preventing their propagation and thus satisfying the second property.

1.5.1 Techniques for tolerating non-malicious arbitrary faults

Non-malicious arbitrary fault types are present not only in any practical distributed system, but in any system that relies on computer components. These faults can be tolerated through error detection techniques, such as integrity checks and semantic checks.

The techniques described below aim to detect data corruption, memory corruption, programmer and operator mistakes. However, each approach mentioned has its overhead cost associated, for either performing repeated checks and recalculations, encrypting, or doubling memory requirements due to redundancy.

1.5.1.1 Integrity checks

Integrity checks verify data by saving at least one redundant data that can be used to validate against the original data, such as checksums, duplicate states, timestamps or data size values. This approach is commonly used when reading and writing data from main memory, storage and peer-to-peer network message exchanges. We now describe in more detail these techniques:

Data and state redundancy (BHATOTIA et al., 2010; CORREIA et al., 2012; SCHWARZ et al., 2004; CLARKE et al., 2003; CHANDRA; GRIESEMER; REDSTONE, 2007): each process variable or stored data has a duplicate which can be validated against and used for backup. The duplicates must be always be kept in sync and checked for consistency on each read and write operation. This approach clearly uses a significant amount of additional memory and has an increased overhead for keeping both states in sync;

Checksums and hashes redundancy (BHATOTIA et al., 2010; CORREIA et al., 2012; CLARKE et al., 2003; SCHWARZ et al., 2004; CHANDRA; GRIESEMER; REDSTONE, 2007): the usage of encoded redundancy allows for future detection of undesired corruption. The most common type of redundancy is generating a checksum or hash of data and attaching it to the protocol messages prior to transmitting across the network or saving them in storage. Any read or write operation on data must recalculate the checksum and verify against the one attached to the message, adding a significant performance cost related to the checksum algorithm used;

Encoding and arithmetic codes (BEHRENS; WEIGERT; FETZER, 2013): the usage of in-place encoding and decoding, like numerical properties of data, can be used to detect undesired corruption in each read and write operation. For instance, if numerical variables are multiplied by a prime number upon writing and divided by the same number when they are read back, the remainder should always be zero. This approach is considered to be very efficient performance-wise, but lacks coverage, as pointed out by (BEHRENS; WEIGERT; FETZER, 2013).

1.5.1.2 Semantic checks

Semantic checks validate that after an operation has been applied on data, the newly obtained state is semantically correct according to the applied operation (BHATOTIA et al., 2010). For instance: after adding an element to a list, check if the element is in the list. This approach has the added benefit of testing the system against possible bugs, which was one scenario in the experiment found in (CHANDRA; GRIESEMER; REDSTONE, 2007).

1.5.1.3 Distributed Validation

In a distributed system in which each replica has its independent state, although total order broadcast can guarantee state transitions are applied in the same order, it cannot guarantee that all replicas will have the same state in the presence of non-malicious arbitrary faults. A replica that experiences an arbitrary fault may have its state diverged from the others, while state transitions will continuously be applied on top of the corrupt state. This may allow the system to display incorrect data when the replica is queried by a client application, or even corrupt the rest of the system if a state transfer mechanism is present.

One possible strategy, experimented in (CHANDRA; GRIESEMER; REDSTONE, 2007), is to compute a checksum of all data in disk and validate if it is the same as other replicas through the network. As pointed out in (CHANDRA; GRIESEMER; REDSTONE, 2007), this method is very resource-intensive to do through serialization, but could be done more efficiently through the underlying storage if it has mechanisms to do so, such as snapshotting or checkpointing. However, this approach does not validate data in memory, it has to assume that all data has been moved to the storage.

1.6 Related Work

Many attempts have been made trying to achieve non-malicious arbitrary fault tolerance through several different approaches. The most common approach observed is to harden a less tolerant fault model towards the most tolerant one by covering each type of fault present in the arbitrary fault class individually. It also seems to be the case where the use of hashes or Message Authentication Codes (MACs) to provide validation checks is one of the most employed techniques used to tolerate several types of faults. In the following paragraphs we detail these approaches.

In (CORREIA et al., 2012), an in-depth non-malicious arbitrary faults study is presented. Many of the techniques presented in this dissertation are inspired by this work. The approach taken was to develop a library that hardens processes built on top of it. All the process' messages, event handlers and variables, if implemented according to the library, are managed by it as part of its state. The library intercepts all messages and event handlers to perform integrity checks on them, and aborts whenever an error is detected. This library is not a middleware, but it can be used to harden existing benign fault tolerant middlewares if implemented on top of the library. Our approach takes an existing middleware and explores the challenges of hardening the middleware itself.

In (BHATOTIA et al., 2010), although it discusses several concepts on tolerating arbitrary faults, it only implements semantic checks. This is similar to part of our approach, however this work has a lower coverage because the checks are implemented only at the

application layer.

The approach presented in (BEHRENS; WEIGERT; FETZER, 2013) involves the use of a low-level encoding compiler so processes read, write and perform all operations with encoded arithmetic values. Whenever a value is changed due to corruption, the arithmetic decoding operation fails and the process detects it. Arbitrary faults handling is mapped to benign faults, so processes either crash or have their messages discarded. This approach also sacrifices coverage for better performance due to the use of arithmetic codes.

In (CHANDRA; GRIESEMER; REDSTONE, 2007), disk corruption is handled by including checksums in data and validating when reading back from persistent memory. Also, a distributed validation mechanism is used, where the storage takes a snapshot, generates its checksum and compares to the other replicas to detect if any of them is faulty, recovering through a state transfer feature. This mechanism restricts itself to data that is present in persistent storage, not addressing main memory corruption concerns, not avoiding propagation or preventing damage to the end-users. In order to address main memory corruption, this work implements the technique of duplicating the database and double-checking on each access.

2 Non-malicious arbitrary fault tolerance through hardening

In the previous chapter we described the two main consolidated fault classes, benign and arbitrary faults, which are not satisfactory for high-scale production environments. We have seen critical systems failing, such as the Amazon S3 2008 outage (CORREIA et al., 2012), and studies confirming the lack of adoption of algorithms for the arbitrary fault model (BHATOTIA et al., 2010). What stands between the two consolidated fault classes is an alternative non-malicious arbitrary fault model that has been gaining a lot of attention in academia (BEHRENS; WEIGERT; FETZER, 2013; CORREIA et al., 2012) and possibly privately by companies.

We chose to explore this fault class further, since we believe the existing research presented in the previous chapter is not complete and the fault tolerance field could greatly benefit from our research results. We use an existing benign Paxos library called Treplica and employ the techniques studied in the previous chapter to harden it towards a non-malicious arbitrary fault model.

2.1 Treplica

Treplica (VIEIRA; BUZATO, 2008; VIEIRA; BUZATO, 2010) is a library coded in Java that allows distributed applications to use Paxos as middleware to manage state replication through its state machine. Its implementation is close to the Multi-Paxos (LAMPORT, 1998; LAMPORT, 2001) approach with a few additional optimizations, like Fast Paxos (LAMPORT, 2006) support and broadcast votes, where each learner agent receiving a majority of votes can commit the change immediately.

In Treplica, replicas can concurrently perform any Paxos role, such as coordinator, proposer, learner and acceptor. This is analogue to many practical middlewares implementing Paxos, and allows for greater flexibility in the amount of replicas and system configurations. Figure 6 illustrates Multi-Paxos algorithm messages exchanged during a common round in a consensus instance, highlighting the differences between theoretical Multi-Paxos and Treplica's implementation. The message roles are as follows:

Message #1: Client proposal message sent to coordinator;

Message #2: Proposal sent to acceptors for voting;

Message #3: Acceptors vote on proposal;

Message #4: Decision is broadcast to learners.

In Treplica, voting messages, labeled #3 in the figure, are received by learners and the proposer as well, thus allowing learners to apply the state transition immediately. Also, the proposer can send the client response, as soon as receiving a majority of votes. Moreover, message #4 is not necessary but is used to broadcast a decision if there is any message loss.

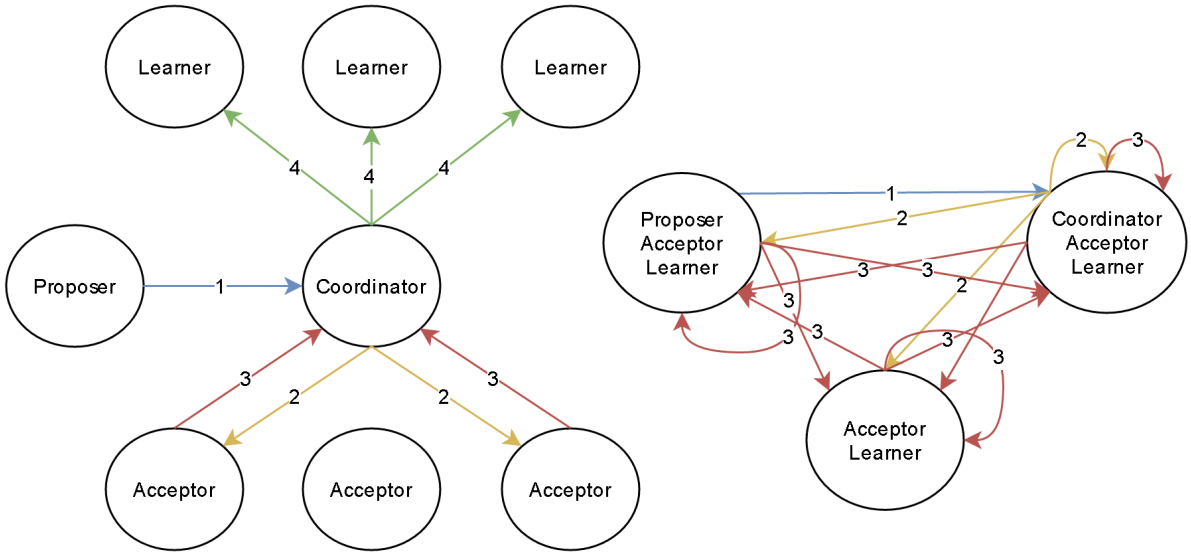

Figure 6: Multi-Paxos (left) and Multi-Paxos in Treplica (right)

Applications designed according to the Model-View-Controller (LEFF; RAYFIELD, 2001) standard can easily be modeled to use Treplica. We chose Treplica because its modular architecture allows for improvements to be easily coded and tested. Since it is designed to tolerate benign faults, upon analysis we validated that it is prone to non-malicious arbitrary faults we are interested in, due to:

1. Reading and writing serialized binary files to the storage, which can become corrupt due to storage faults;
2. Usage of UDP protocol for serialized message exchanges, which can be corrupted on noisy network channels;
3. Usage of Java virtual machine, which can have its process memory space corrupted at runtime.

Additionally, Treplica is object-oriented and makes use of immutable objects design, where an object is never changed after being instantiated. This allows for more efficient use of checksums. State transition semantic checks can also be easily coded by the application due to its integration with the state machine modelling.

2.2 Tolerating non-malicious arbitrary faults

We decided to create a set of validation techniques to harden the Paxos algorithm, looking for the ones that best match the software architecture of Treplica.

Our main approach to harden our existing benign crash-recovery fault model towards the non-malicious arbitrary one is to employ error detection techniques and a distributed validation mechanism to detect errors resulting from data corruption, while initially not worrying about how to recover from them. Upon detecting the errors, our proposal is to abort the replica execution, preventing it from propagating corrupt data and further participating in the algorithm. This strategy reduces our existing crash-recovery fault model to a simpler crash-stop one, but tolerating non-malicious arbitrary faults. Our middleware library of choice does not currently have the state transfer mechanism that would allow us to achieve the crash-recovery fault model tolerating non-malicious arbitrary faults.

From the point of view of a benign fault model distributed system, most arbitrary faults behave as silent faults because their errors cannot be detected. For instance, if a user clicks a button to buy one book, but a replica processes that two books have been bought because bits got flipped along the way, then this is not an error from Paxos point of view, because the message was delivered consistently across all replicas. In order to effectively tolerate such silent faults, we employed the following techniques:

- Integrity checks, to address data corruption.
- State checks, to address state corruption.
- Semantic checks, to address programmer mistakes.
- Distributed validation, to address main memory corruption breaking the Paxos algorithm.

2.2.1 Integrity checks

We often check for data corruption as soon as it can be detected. Whenever an immutable Paxos message object is instantiated, either to be written to persistent storage or propagated to the network, we calculate a checksum of its contents and append to it. When the message is received or recovered from storage, the checksum is recalculated and validated by comparing it to the one attached. We acknowledge that recalculating a hash every time some data is read adds overhead, but Treplica's modular architecture allowed us to identify key locations in the source code to minimize overhead. Through this implementation we should be able to detect any corruption that affects messages, such as network messages, persistent storage and main memory corruption.

Figure 7 illustrates what each replica does during a regular message transmission in order to detect data corruption. In Algorithm 1 it is shown that any message object now inherits from AuthenticatedObject class, which provides abstract methods that should be implemented, such as generateHash(), and a hash attribute used to store the generated hash. The Utils class has the implementation that generates the hash, while it expects the content to be hashed to be supplied in its static method Utils.hash(). Finally, to authenticate the message upon receiving it, all that is necessary to do is generate the hash again through the implemented abstract method and compare it to the hash attribute.

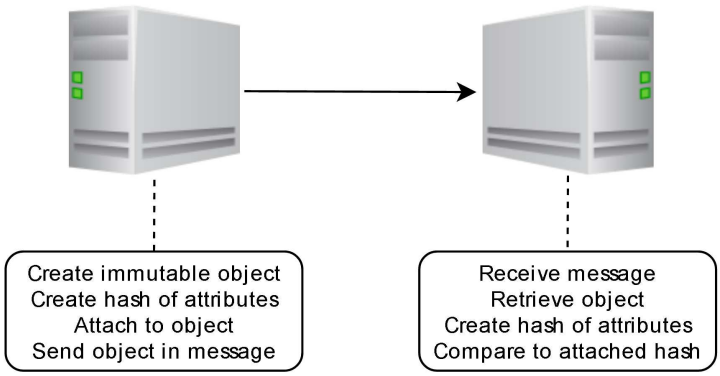

Figure 7: Data transmission between two replicas with data corruption validation

Algorithm 1 Data corruption check

```
class ObjectInstance extends AuthenticatedObject
        implements Serializable {

    ObjectInstance(String attr1, String attr2){
        this.attr1 = attr1;
        this.attr2 = attr2;
        this.hash = generateHash();
    }

    void generateHash(){
        return Utils.hash(this.attr1, this.attr2);
    }
}

void processMessage(){
    AuthenticatedObject message = queue.dequeue();
    assertEqual(message.hash, message.generateHash());
}
```

2.2.2 State checksum

In order to validate the application state, we first needed to define a state checksum. We inspired ourselves on Git version control system (TORVALDS, 2005), where each commit has a unique hash. The state checksum is generated after each successful state transition, and is calculated using the previous state checksum, the current state transition data and the current state information. The first step is to obtain the state information. To perform this step we employ two alternative ways:

Serializing the application state: This approach consists in serializing all the state data. It may be very costly for applications which have their state constantly growing, even if linearly. For applications with constant-size state, this approach can be advantageous and offer full coverage. We call this approach "deep check" in the code.

Application-defined state information: This approach delegates to the application the task of defining the state. By implementing an interface method, the application must return a data-set that contains significant information to represent a state. For some applications, this may be as simple as the number of entries in a data structure.

Once the state information is obtained, the state checksum is calculated and replaces the previous state checksum. In Figure 8 it is shown what the state checksum consists of and how it is replaced by a new one. The stateCount attribute is an integer identifier of the state, where the hash attribute stores the hash that represents the state information, and is used to generate the next one. The Algorithm 2 shows that whenever a new StateChecksum object is created, the previous state checksum's hash, the current state hash and the actions that caused the state transition are used to generate the new state checksum. Algorithm 2 also shows that the state checskum is updated whenever a trasition action takes place. The condition that evaluates if the attribute deepCheck is enabled decides which alternative way the state information is obtained.

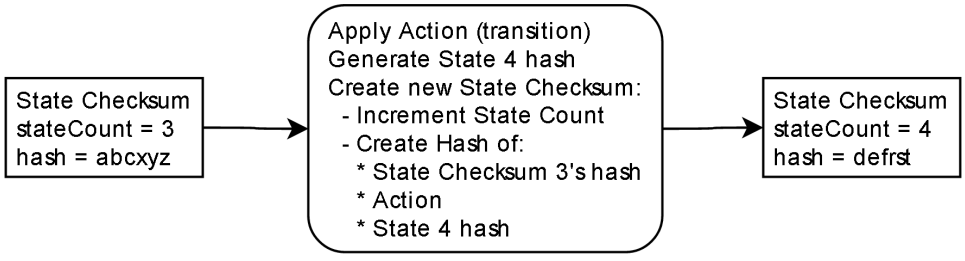

Figure 8: State checksum generation steps

Algorithm 2 State checksum generation

```
class StateChecksum {

    StateChecksum(StateChecksum previous,
            byte[] currentHash, Action action){

        if (previous != null){
            this.stateCount = previous.stateCount + 1;
            this.hash = Utils.addHash(previous.hash,
                    currentHash, action);
        } else {
            this.stateCount = 0;
            this.hash = new byte[0];
        }
    }
}

class StateMachine {

    void processAction(Action action){
        ...
        byte[] stateHash = byte[0];
        if (this.deepCheck){
            stateHash = Utils.generateHash(this.state);
        } else {
            stateHash = this.state.getStateHash();
        }
        ...
        Application.currentChecksum = new StateChecksum(
                application.currentHash, stateHash, action);
        ...
    }
}
```

2.2.2.1 State checks

We attempt to detect application state corruption by having a duplicate state and comparing differences between it and the main state of the application. Every time a state transition takes place, we apply the state transition operation to both states, then we generate and compare their state checksums. This allows for any state transition operation that silently fails and causes the states to diverge to be detected before any error propagates. Both states are also validated every time the application state object is requested, since they can become corrupt at any time due to memory corruption.

Figure 9 shows that a state has two copies, and each action is first applied on both states and then their integrity is validated. In Algorithm 3 we can see that a hash of each

state is obtained after each state transition, the validation occurs by ensuring that both hashes are equal. Finally, when reading the state, both hashes are obtained again and tested for equality.

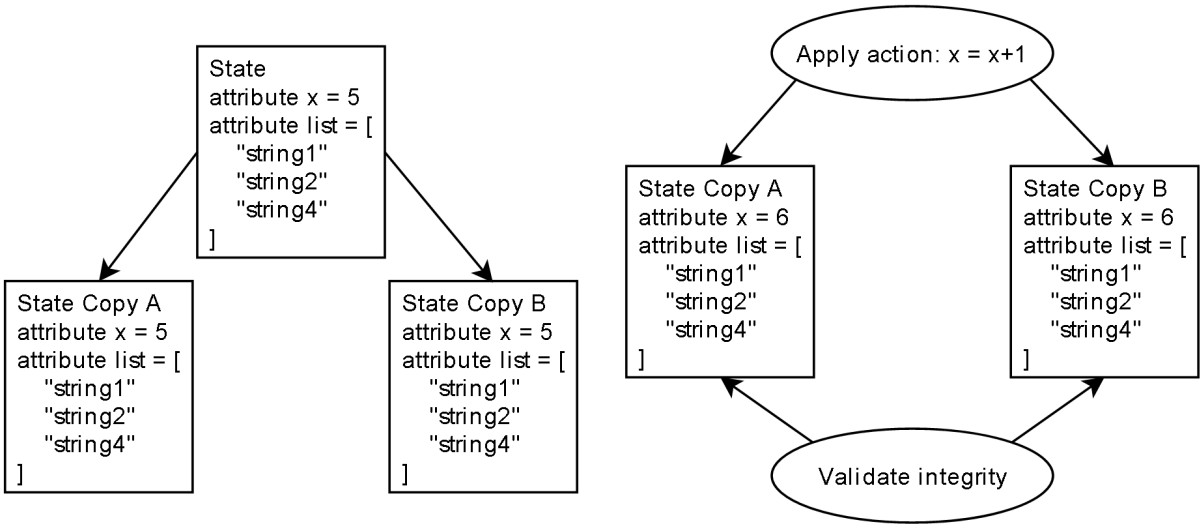

Figure 9: State duplication and state integrity checks

Algorithm 3 State integrity check

```
class StateMachine {

    void processAction(Action action){
        ...
        Serializable resultA = action.executeOn(this.stateA);
        Serializable resultB = action.executeOn(this.stateB);
        byte[] stateHashA = this.stateA.getStateHash();
        byte[] stateHashB = this.stateB.getStateHash();
        assertEqual(stateHashA, stateHashB);
        ...
    }

    State getState(){
        byte[] stateHashA = this.stateA.getStateHash();
        byte[] stateHashB = this.stateB.getStateHash();
        assertEqual(stateHashA, stateHashB);
        return stateHashA;
    }
}
```

2.2.3 Semantic checks

We introduced semantic validation as an additional way to detect main memory corruption and programmer mistakes. For each state transition operation implemented by

the application, it is required to implement a semantic validation method that verifies if the transition has been correctly applied to the state. This semantic validation method is run as soon as the state transition is applied, thus if the validation fails, all further operations on the given replica are halted.

In Figure 10 we have a simple example state transition that adds a string element to a list and sorts it. The semantic check consists in checking if the element is in the list and if it is sorted after applying the state transition. In the Algorithm 4 it is shown that the semantic check has to be implemented in the action itself. The InsertAction class is an implementation of Action interface that has the state transition operation implemented and its semantic check.

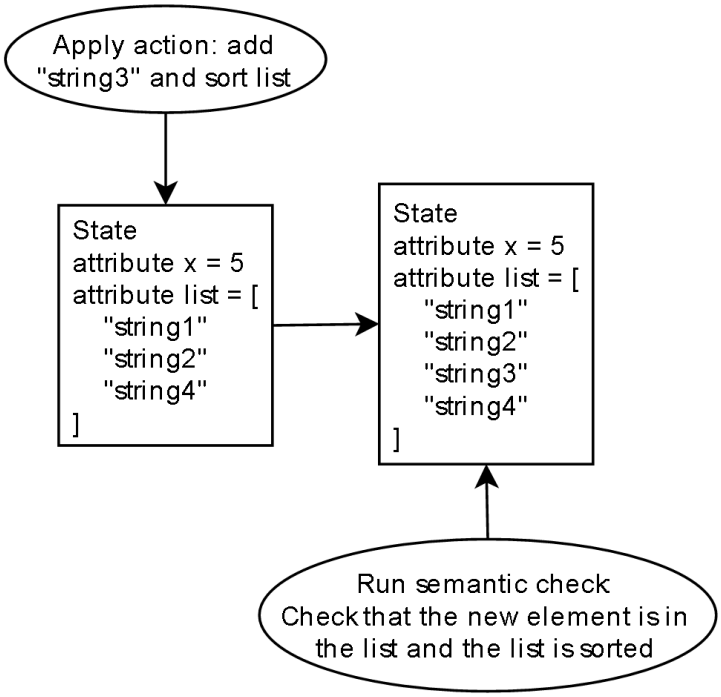

Figure 10: Semantic validations

2.2.4 Distributed validation

Besides state corruption, the Paxos algorithm can also be affected by arbitrary faults, thus it may cause wrong votes to be cast in the rounds, leading to incorrect state transitions being applied, resulting in a replica with a diverging state. In order to completely satisfy the "No propagation" property of the fault model, this scenario must be covered by a validation mechanism.

A distributed validation mechanism can allow replicas to validate their state upon receiving a network message containing a checksum or hash that is related to the state they currently are, thus detecting possible state divergences. We developed a way to use

Algorithm 4 Semantic check

```
class InsertAction implements Action {

    InsertAction(int x){
        this.element = x;
    }

    Serializable executeOn(State state){
        List<String> list = (List<String>) state;
        boolean result = list.add(this.element);
        return result;
    }

    void semanticCheck(State state){
        result = true;
        result &= state.list.contains(x);
        result &= isSorted(state.list);
        assertTrue(result);
    }
}

class StateMachine {

    void processAction(Action action){
        ...
        Serializable result = action.executeOn(this.state);
        action.semanticCheck(this.state);
        ...
    }
}
```

the Paxos algorithm to perform this validation, thus having the algorithm extended with this mechanism.

We attempt to detect state divergencies between replicas by including the state checksum in the voting messages exchanged by Paxos in the accepting phase (see section 1.3.1). Acceptors read the checksum from the application when creating the immutable voting messages and attach it to them. In Treplica, all replicas receive the voting messages, thus the learner module validates the local state upon receiving them using the attached checksum.

In order to minimize the performance impact and adapt the mechanism to Treplica's architecture, we decided to take an eventual and opportunistic validation approach. By defining a window of state counters in which the state checksum is updated, the replicas are able to eventually validate a state within the defined window. For instance, if the

window value is "100", then the state checksum will change only every one hundred state transitions have been applied. This makes it easier to synchronize all replicas in the same window. In Figure 11, acceptors include their state checksum numbered #14 when consensus instance of the state transition numbered #15 is running. Both consensus instances #14 and #15 are related to the same window, which is from state transition #1 to state transition #100, thus they carry the checksum generated in transition #1. The learner validates state checksum numbered #14 before committing the state transition numbered #15. In this example, if any of the replicas have their state diverging within this window, it will be detectable only after transition #100, where a new checksum will be included in exchanged messages. Without this window mechanism, replicas would rarely be able to validate received state checksums related to the same state count, because they apply the state transitions in an asynchronous way. Treplica packages varied amounts of state transitions in the same Paxos instance, and replicas end up advancing rounds in different paces, resulting in the current window and backlog variables frequently getting discarded due to the state count advancing before having a chance to validate.

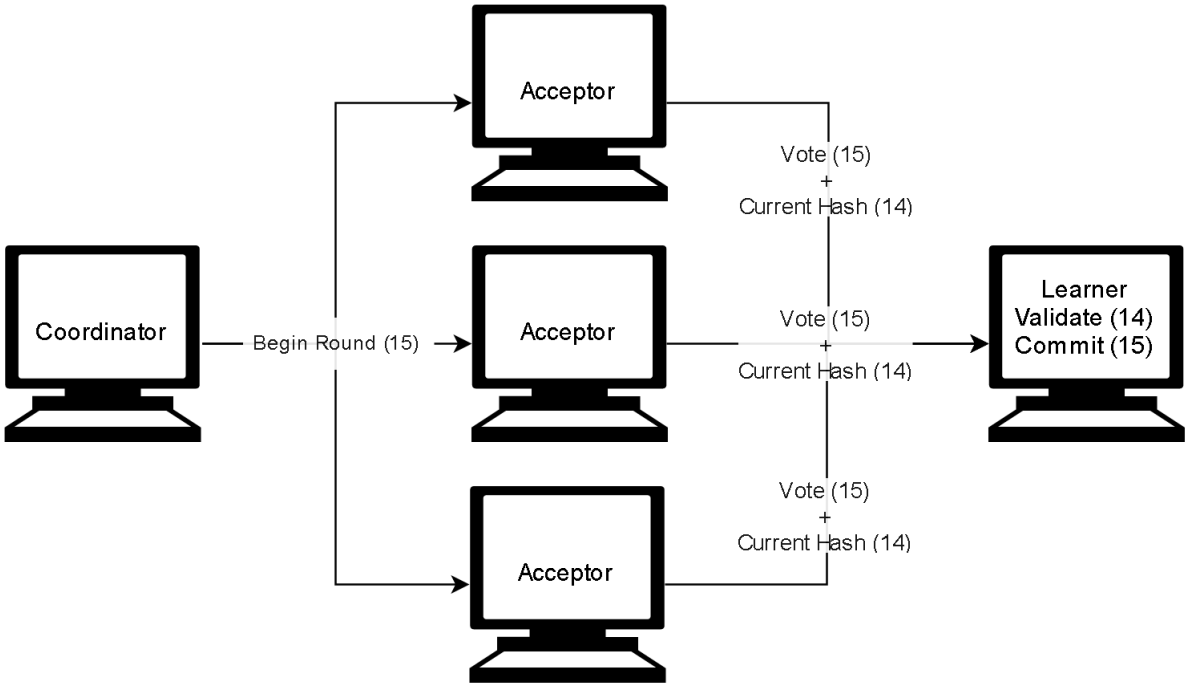

Figure 11: Distributed validation mechanism implemented in Treplica

We defined two variables where we store the received state checksums. One that is related to the current window, and a backlog one that is related to the next window to be processed. The next window is determined by the first received message that does not fit into the current window. Messages received that fit into the current window are validated immediately, while messages related to the subsequent registered window are stored for later validation. When a replica advances to the window that contains messages to be

processed later, it moves all the stored messages to the current window variable, clears the backlog one and starts processing them.

Validation consists in comparing the validating replica's state checksum to the majority of received state checksums. As soon as the number of received common state checksums is the same as the number of replicas in the majority quorum in the system, the validating replica's state checksum is compared to this common state checksum. If the validating replica's state checksum is not the same as the majority, then the validating replica detects that it has diverged and aborts execution.

In Algorithm 5 it is shown the validation code that is performed by learners. Every state checksum received that is not related to the current window is saved for later processing. When a state checksum that matches the current window is received, it is saved in a structure responsible for storing state checksums indexed by replica unique identification numbers. This structure has a method getMostCommonChecksum() responsible for returning a list of the most common occurrence for the current window. If the size of this list matches the quorum size, then the learner validates its own checksum against that common checksum, which raises an exception if it diverges.

Algorithm 5 Distributed validation

```
void receiveVotingMessage(Message message){
    StateChecksum checksum = message.stateChecksum;
    if (checksum.stateCount == Application.currentChecksum.stateCount){
        saveAndProcessStateChecksum(checksum);
    } else {
        if (backlog.stateCount == checksum.stateCount){
            saveMessageInBacklog(checksum);
        }
    }
    processVotingMessage(message);

}

void saveAndProcessStateChecksum(StateChecksum checksum){
    currentWindow.saveMessageInCurrentWindow(checksum);
    List<StateChecksum> list = currentWindow.getMostCommonChecksum();
    if (list.lengh == Application.quorumSize){
        if (checksum.hash != list.get(0).hash){
            throw new StateDivergedException(
                    Application.currentChecksum);
        }
    }
}
```

We chose to have the validating replica aborting execution when it detects it has

diverged from a majority because it is the only guarantee we have that the validating replica is the one that diverged. Also, we expected to have it recovering through a state transfer mechanism, but we currently do not have such feature available.

We consider this mechanism to be eventual, since replicas may not participate in certain voting rounds, and would fail in case a majority of replicas diverge at the same time, which we consider to be a unlikely practical scenario if the cluster comprises of more than 3 replicas. Such mechanism would still depend on the application being able to generate a checksum of its state or of certain data that is comparable to other replicas. The more precise this information, the more coverage this mechanism can achieve.

2.3 Implementing the chosen techniques

The collection of techniques described in the previous sections was implemented in Treplica without major difficulties. Treplica's architecture is very modular and layered, and the techniques themselves were adapted to fit accordingly. There were some cases where some existing Treplica layer boundaries had to be broken, for instance, where the learner module had to access a reference to the application layer in order to retrieve the application state checksum to perform the distributed validation.

There were a few new configuration variables introduced along with the techniques:

Hash generation types: We decided to have CRC32 (32-bits Cyclical Redundancy Check) (PETERSON; BROWN, 1961) and SHA-2 (Secure Hash Algorithm 2) (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST), 2002) hash types supported in order to analyze performance impact and coverage differences between both types.

State checksum generation types: We also introduced a configuration option to enable or disable the two approaches of state checksum generation described in the previous section, one that would serialize the state and other that would use the application-defined state information.

A new "Authentication" class was created to manage the new techniques introduced. This class was responsible for providing the logic behind generating hashes and checksums (using CRC32 or SHA-2), interface methods to be implemented by existing classes which were modified to support the validation techniques, and a hub for some static variables that needed to be accessible from all layers, such as the references to the current state checksum in the application layer.

3 Experimental validation

In this chapter we validate whether the hardened Treplica is more fault tolerant than the unmodified Treplica and the performance impact of such hardening. First and foremost, we needed to create the occurrence of non-malicious arbitrary faults in our test system to be able to analyze their resulting errors on the original unmodified Treplica and detect them on the hardened Treplica. Our approach to do so was through randomly generated fault injections. Upon validating the coverage of our implementation, we proceeded to measure the performance impact. We have compiled a set of runs that allows us to compare each variation of the implementation between themselves and the unmodified Treplica, thus concluding whether our approach is viable.

3.1 Testing framework

Our testing framework consists in generating load to simulate one thousand requests per second to each replica in our test system. Each replica runs an instance of an example application on top of Treplica. First, we generated load with fault injections, measuring the detection ratio of the hardened Treplica and the failure ratio of the unmodified one. We then proceeded to generate load without any fault injected, and measure the performance impact of the hardened Treplica versus the unmodified one.

In order to generate the load, the test system running an example application needs to receive requests which cause state transitions for five minutes. We created a manager application to run several cycles of the same test, preparing and cleaning each replica instance resources used by each cycle, such as storage folders and logs. When starting a test cycle the manager application starts the replica instances and starts the load generating application, which sends the requests. In case an error is detected, the application crashes or times out, then the manager application aborts the test cycle execution, cleans the storage folders and logs the results to a separate folder indexed by the test cycle count. Our request timeout was set to five minutes, in order to detect whether a failure made the system unable to continue processing requests.

3.1.1 Test applications

We coded two example applications built on top of Treplica to perform tests on:

Hashset of strings: client requests can add, remove and list elements in a hashset of strings. This example application allows us to measure performance indicators

related to applications that have a growing state, and analyze hardening coverage in situations where any value from any element can become corrupt. In this application, the application-defined state information (see Section 2.2.2) implemented retrieves the number of elements in the hashset. The semantic checks (see Section 2.2.3) implemented for the "AddElement" operation validates if the added element is present in the hashset, while for the "RemoveElement" operation it validates if the element is not present in the hashset;

Single value application: client requests can increment, decrement and display a single integer counter value. This example application allows us to measure performance indicators related to applications that do not have a growing state. In this application, the application-defined state information implemented retrieves the value itself. No semantic checks were implemented for this application. This example application was used only for performance tests.

Each request was always an "Add element to hashset" operation for the hashset of strings application, where the element consisted of a Long-type counter value converted to string value, appended by the replica identification number. For the Single-value application, each request was always a "Increment value". When running tests with fault injections using the hashset of strings application, we always printed all the elements in the hashset when the test finished without interruptions to get a record of the final state of the replica.

3.1.2 The test system

Instead of running Treplica on actual replicas, we opted to run three instances of the same application at the same time in a single machine. Each instance had a separate folder for individual storage. The experiments were executed in a virtual machine running in a personal computer test system which comprised of:

Host CPU: Intel i7 3820 3.6 GHz quad-core with Hyper-threading (Turbo-boost and SpeedStep disabled);

Host RAM Memory: 12 GB DDR3 2133 MHz;

Host Storage: 2 TB Western Digital Caviar Green HDD, 256 GB Samsung 840 Pro SSD;

Host Operating System: Windows 7 Home Premium (installed in SSD);

Virtualization software: Oracle VirtualBox 4.3.24

Virtualized Operating System: Linux Mint 16 "petra" XFCE 32-bits

Virtualized CPU: 4 cores, using hardware virtualization extensions

Virtualized RAM Memory: 4 GB

Virtualized Storage: 25 GB (installed in HDD)

Java version: Java Development Kit (JDK) and Java Runtime Environment (JRE) version 1.7.0 update 51

All performance tests were executed while the host operating system was idle, with no background services running. The specific amount of requests per second we chose was calibrated in the test system described above, to not let the application be limited by processor, memory and storage performance.

3.2 Fault Injection using AspectJ

In order to test our implementation through fault injection, we used an aspect-oriented library known as AspectJ. It allows us to change the behavior of any Java program without changing its main code. In Algorithm 6 it is shown an example injection, where in order to inject a fault in the operation of adding a string to a list, a method must be created to have its behavior overridden. The method in this case is "listAdd". Our injection code runs instead of the original code every time "listAdd" is invoked, we then use a local variable to decide whether we inject a fault that consists in running the original function with a different argument value, or we allow the function to continue without faults.

Algorithm 6 Example injection

```
private List<String> list;
boolean inject;

boolean listAdd(String name){
    list.add(name);
}

boolean around(String name) : listAdd(name){
    if (inject)
        return proceed(random.nextLong().toString());
    else
        return proceed(name);
}
```

Our technique was to generate corruption errors through byte flips and value changes that would attempt to pass undetected through our validation techniques implemented as discussed in the previous chapter. The fault injections coded are compatible with both the hardened Treplica and unmodified one, so we could easily compare them.

3.2.1 Injection framework

We created a fault injection framework that reads a configuration file when Treplica is initializing and sets the fault injection conditions and modes according to what is defined in the configuration file. Whenever an aspect-marked method in the original code is executed, the aspect code is invoked. We implemented every aspect to first verify the injection conditions configured for the given injection before injecting any fault. If the condition is not met, no code is injected and the original code is run. The two modes of operation implemented are:

Single timed injection mode: in this mode, a timer value is defined in the configuration file for each injection. Once the timer expires, the condition allows the fault to be injected. The fault is injected the next time the aspect is invoked. Once the fault is injected, the condition is permanently disabled. This mode is useful when we want to confirm that a single fault injected caused a single error, and whether the error was detected or not;

Probability-based injection mode: in this mode, a probability value is defined in the configuration file for each injection. Whenever the aspect-marked code is executed, it randomly generates a probability value. If it is higher than the probability value defined, then the condition for fault injection is met and the fault is injected immediately. All subsequent executions of the aspect-marked code will check against the probability of injection, which may inject more fault occurrences. This mode is useful when we cannot guarantee that a single fault injection can cause a failure to occur, because it may require specific conditions to be met. For example, a failure may require many consecutive messages to be lost, else the errors will be absorbed by the algorithm.

3.2.2 Injection scenarios

- For message injections, we used single timed injection mode to corrupt middleware protocol messages, such as the ones used for network communication between replicas, log of operations saved and retrieved from storage, and checkpoints, also saved and retrieved from storage. The injection consisted of:

  Message injections: In network, storage and checkpoint messages, we change a random value in the message as soon as it is received from the network or recovered from storage, while retaining the checksum value on the hardened Treplica.

  Consequence in original Treplica: Depending on the message value changed on fault injection, the middleware would crash by reading an unexpected

value, hang indefinitely making the application become unresponsive, or apply the incorrect transition operation. Both the availability and reliability properties would be compromised.

Consequence in hardened Treplica: The error from the protocol message fault injection would be detected before processing the message.

- For application injections, we used single timed injection mode to inject faults during state transition in the hashset of strings example application. The injection consisted of:

  Application injections: We inject faults in the state by manipulating the state transition operation to either not add the elements or by changing their string values.

  Consequence in original Treplica: The failures would only be noticeable upon querying the application data. The middleware would continue to work without problems. The reliability property would be compromised, but the availability property would not.

  Consequence in hardened Treplica: The resulting state checkums generated on the duplicated states would end up being different depending on the state information approach used, and thus allowing the error to be detected. Semantic checks, if present and implemented accordingly, would also be able to detect such errors.

- For Paxos injections, we used probability-based injection mode attempting to cause failures by breaking simple but very important algorithm mechanics, in order to cause replica states to diverge. The faults injected are as follows:

  Learner commits with no quorum: This injection causes the affected learner to have a probability of committing a proposal without requiring a quorum to do so. If the round does not succeed and a new value is proposed, the replica ends up with its state diverging from the others.

  Acceptor forgets its previous votes: This injection causes the affected acceptor to have a probability of returning no previously registered votes for a given round. The coordinator, upon receiving this message, will assume that no previous value has been proposed, and will propose a new one instead, while the previous value may have been decided and committed by learners.

  Coordinator forgets last proposals received: This injection causes the coordinator to have a probability of forgetting the previous proposals he receives from acceptors, thus choosing to start a new round with a new value while the previous value may have been decided and committed by learners.

Test	Runs	Fault injections	Error detections	Failures	Rate
Message injections - Fault on a protocol message received	100	100	0	100	0%
Application injections - Fault when adding element	100	100	0	100	0%

Table 2: Injection tests performed on original unmodified Treplica

The items below are the same for the three injection scenarios above:

Consequence in original Treplica: The failures would only be noticeable upon querying the application data, printed at the end of the test runs. The middleware would continue to work without problems. The reliability property would be compromised, but the availability property would not.

Consequence in hardened Treplica: All scenarios would cause at least one replica to commit an undecided proposal, causing a state divergence. The divergence would cause the state checksum generated on the diverged replica to be different than the others, thus the divergence would be detected by the distributed validation technique. If a majority of replicas diverges at the same time, then the failure would not be detected, breaking the distributed validation mechanism permanently.

An example work flow of a state transition from the receiving of a Paxos message to the client response showing the differences between the unmodified and the hardened Treplica versions, with fault injection, can be seen in Figure 12.

3.2.3 Injection test results

In Table 2 it is displayed the tests executed in the original unmodified Treplica using timed injection mode. We registered failures such as the application becoming unresponsive, crashing or displaying incorrect data in every test run. Since the faults injected were random, message injections were able to cause any type of failure. As for application injections, all of them only caused incorrect application data to be displayed.

Table 3 lists the execution of fault injection tests in hardened Treplica using timed injection mode. We confirmed the detection of all error occurrences either through logging or replicas aborting their executions upon detection.

As for Paxos injections, shown in Table 4, we used probability-based strategy to inject faults. We used 80% fault injection probability for the injection scenario being tested, while having 20% injection probability of message loss at the same time. We needed to simulate a scenario with message loss in order to create the possibility of the injection scenario to cause some failures. For instance:

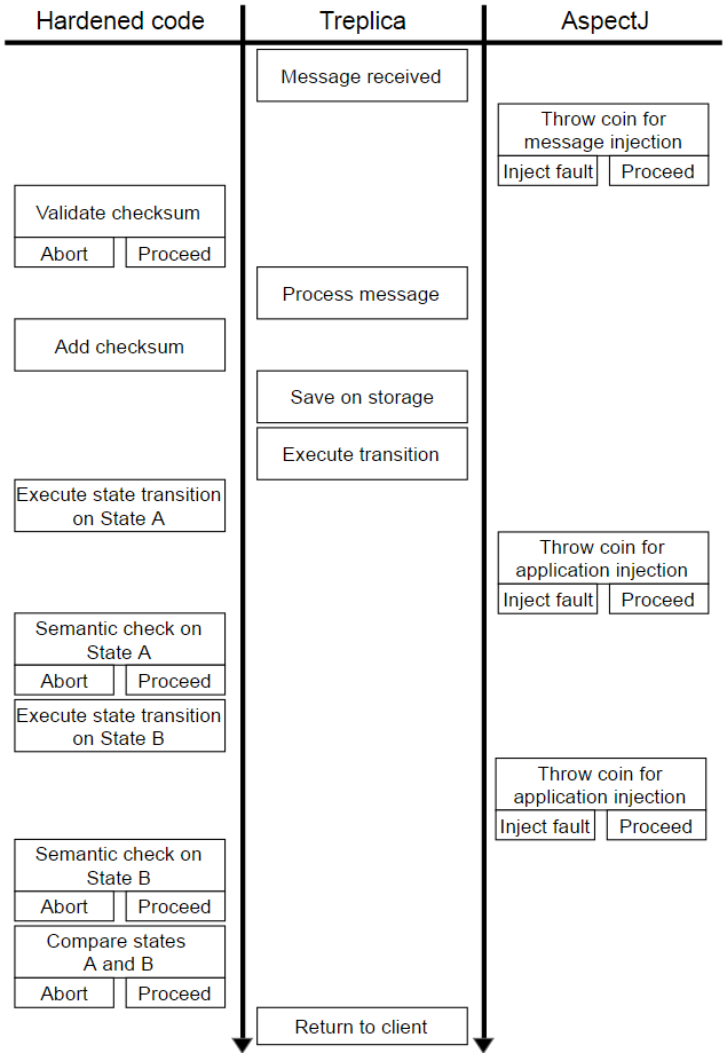

Figure 12: Transition execution with fault injections in the hardened Treplica

Test	Runs	Fault injections	Error detections	Failures	Rate
Message injections - Fault on a protocol message received Captured by: Hash check	100	100	100	0	100%
Application injections - Fault when adding element Captured by: Semantic check, Duplicate state comparison	100	100	100	0	100%

Table 3: Injection tests performed on hardened Treplica

Test	Runs	Fault injections	Errors detections	Failures	Rate
Learner commits with no quorum Single replica	50	22558	41	1	98%
Acceptor forgets previous votes Single replica	50	36129	40	0	100%
Coordinator forgets received proposals Single replica	50	4710	12	6	88%
Learner commits with no quorum All replicas	50	65646	47	3	94%
Acceptor forgets previous votes All replicas	50	42950	49	0	100%
Coordinator forgets received proposals All replicas	50	6855	42	8	84%

Table 4: Paxos injection tests performed on hardened Treplica

- The learners need message loss to commit a proposal which has not been decided;
- The acceptors need message loss to send the coordinator an invalid last voted proposal, while at least one replica has already committed during the previous, supposedly failed, voting round;
- The coordinator needs a message loss to restart a failed voting round and propose an invalid value, while at least one replica has already committed in the previous voting round.

For the test runs where errors were not detected, there were cases where there were no failures, thus there was no state divergence between any of the replicas. We confirmed such scenario by comparing all the elements printed by each replica at the end of a test run. The runs where there were failures, however, a majority of replicas diverged to different states, rendering the distributed validation incapable of detecting any divergence for the rest of the test run. These cases could also be confirmed by comparing the printed elements. We consider these occurrences to be beyond our technique's coverage capability at this moment.

For the purpose of experimenting with more faults, since our injections for this scenario were probability-based, we also ran tests where we performed Paxos injections in all replicas. Those test runs are labeled "All replicas" in the table, whereas the "Single replica" labels refer to injections performed in a single replica. We noticed that the test runs where we performed injections in all replicas were more likely to result in all replicas diverging. Additionally, we analyzed the possibility of the fault injected causing failures only to the replica which it was injected, or if errors propagated. In all our test runs, at least one execution had error propagation.

We did not run any Paxos injection test scenario in the original unmodified Treplica because we already observed in the hardened Treplica that the failures do not compromise

the availability properties of the algorithm. These failures can be mapped to the application tests we performed in the original unmodified Treplica, affecting the application state that is only noticeable by the client, compromising the reliability property. The probability of these injections not causing any failure is already measured in the hardened Treplica test runs. Thus, the rate of error detection in the unmodified Treplica is 0% in all Paxos injection scenarios.

3.2.4 Injection results discussion

By executing the injection scenarios in the original unmodified Treplica, we were able to confirm that reliability and availability properties were compromised. By injecting faults in protocol messages, the replicas would not only display benign faults such as lock-ups and crashes, but also perform incorrect application operations, vote in incorrect rounds, and flood the network with invalid round messages. Some faults injected did not affect Paxos, but they significantly affected the application state consistency, being noticeable by the client. Our hardened Treplica however, was able to detect errors in all the fault injection scenarios, except for when all replicas got their state diverged in Paxos injection scenarios. Through fault injection we were also able to detect bugs in our implementation that did not appear during development.

Diagnosing errors was a challenge on its own. Storage corruption resulted on the same replica failing over and over because it was reading corrupt middleware protocol data or checkpoint every time it restarted, rendering the replica incapable of recovering from this fault unless a state transfer mechanism was implemented. A corrupt network message could be simply dropped. All middleware protocol faults could be mapped to benign faults and any error propagation prevented.

Main memory corruption was found to be the most difficult to cover, test and diagnose. Our approaches of performing semantic checks and generating a checksum to be compared later with the duplicate application state and distributed validation were able to cover main memory corruption affecting the application state, but the coverage was only as good as the implementation of the semantic checks and state information methods. All application state corruptions were mapped to benign faults by aborting the replica execution upon detection. When the replica is restarted, the state transitions can be reapplied, with a very high chance of avoiding the previous occurrence of memory corruption.

Even though we wanted to address overall memory corruption, we did not entirely cover internal middleware state corruption affecting Treplica itself. We covered some middleware corruption scenarios with a few Paxos fault injections we believed could have errors propagated. During our tests we confirmed that our three Paxos fault injections scenarios propagated errors and caused other replicas to have their state diverged, instead

of only the one the fault was injected into.

From possible main memory corruption occurrences, we list below two types of faults that we did not cover and could have errors propagated:

1. The highly unlikely case of data corruption in main memory between instantiating an immutable object and generating its checksum (or generating an incorrect checksum), for this scenario we are not taking any action;

2. Main memory corruption in internal Paxos state, which can lead to erroneous behavior, like a replica getting lost between voting rounds, voting incorrectly or the coordinator starting invalid voting rounds. Our analysis indicates that using hashes for this type of validation would degrade system performance greatly, thus we considered a different approach, using the distributed validation mechanism. Any error in Paxos causing a replica state to diverge can be detected, as long as they do not happen on a majority of replicas in the same state count window. Although, we do not consider our distributed validation mechanism an approach to increase coverage on main memory, since it did not prevent propagation. Its strengths are aimed at detecting state corruption, programming and configuration mistakes.

The only scenario we injected faults and we could not detect errors was the case of a majority of replicas (two replicas in our test suite), diverging while in the same state count window, thus leading to a failure. We consider the following conditions are needed for this to happen:

- The smaller the amount of replicas in the cluster, the higher the probability. In our test suite there were only three replicas, and we experienced this case in 6% of all test runs. With more replicas, the divergence can be detected before a majority of replicas is compromised;

- There must be several consecutive messages lost while the fault is injected. Our test runs had 20% of message loss in all replicas to induce the failure. We validated that without message loss, there could not be any failure.

- The majority of replicas needs to diverge within the same state validation window (see Section 2.2.4). The divergence of individual replicas can be easily detected once the validation window changes and updates the state checksum, if there is still a majority of correct replicas.

We summarize our test suite results and analysis in Table 5, where we check for each injection scenario we experienced:

Fault	Detected?	Fault model mapped to	Can be propagated?	Rate
Message injections (network)	Yes	Benign crash-recovery (single message loss)	No	100%
Message injections (stable storage)	Yes	Benign crash-stop (replica unavailable)	No	100%
Application injections (memory corruption)	Yes, but relies on injection, semantic check implementation and state information method implemented by application	Benign crash-recovery (replica restarted)	Not applicable	100%
Application injections (bugs)	Yes, but relies on injection and semantic check implementation	Benign crash-stop (replica unavailable)	Not applicable	100%
Paxos injections	Mostly	Benign crash-stop (replica unavailable)	Yes	94%

Table 5: Non-malicious fault class coverage

- If the error from the fault injected was successfully detected and which conditions were necessary for this detection;
- To which fault model the fault occurrence can be mapped to and what other replicas observe of the faulty replica;
- If the error can be propagated and disrupt other replicas;
- Our coverage rate for the given fault.

According to our analysis, we believe that upon detecting the errors, the most effective approach to recover a replica from most arbitrary faults is to transfer a fault-free state from another replica, resetting the replica to a pristine state in the distributed system's state machine. However, our solution so far has been to abort the replica execution because Treplica currently does not have a state transfer feature implemented. Our current solution results in a crash-stop non-malicious arbitrary fault model instead of the crash-recovery non-malicious arbitrary fault model we initially intended to achieve. There is only a limited number of faults we can recover from while we do not have state transfer feature available. Figure 13 displays what fault tolerance model we achieved in our experiment, where we tolerate with benign faults in the crash-recovery fault model, and tolerate non-malicious arbitrary faults in the crash-stop fault model.

We consider the non-malicious arbitrary crash-stop fault model to be more resilient and more practical than the original benign crash-recovery implementation. If a benign fault occurs, the system is able to recover itself and continue, but if an error from a non-malicious arbitrary fault is detected and is non-recoverable, we abort the replica execution, preventing any propagation of erroneous behavior.

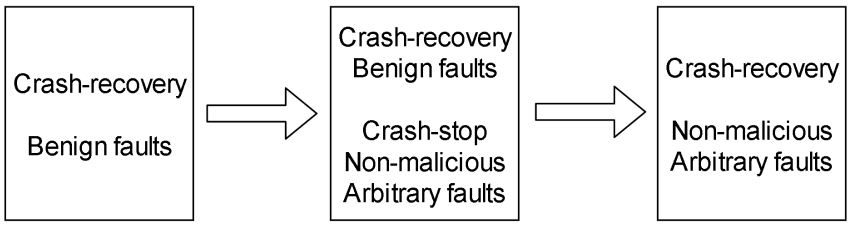

Figure 13: Hardening of fault models

Example Application	Encoding	Complete serialization	Message Length	Legend
Hashset	CRC	Yes	Normal	HS_CD
Hashset	CRC	No	Normal	HS_CN
Hashset	CRC	No	Null	HS_CN_NM
Hashset	SHA-2	Yes	Normal	HS_SD
Hashset	SHA-2	No	Normal	HS_SN
Hashset	SHA-2	No	Null	HS_SN_NM
Hashset	None	Not applicable	Not applicable	HS_OLD
Single-value	CRC	Yes	Normal	SV_CD
Single-value	CRC	No	Normal	SV_CN
Single-value	CRC	No	Null	SV_CN_NM
Single-value	SHA-2	Yes	Normal	SV_SD
Single-value	SHA-2	No	Normal	SV_SN
Single-value	SHA-2	No	Null	SV_SN_NM
Single-value	None	Not applicable	Not applicable	SV_OLD

Table 6: List of performance test parameters

3.3 Performance tests

In order to measure the performance impact of our implementation, we compared fourteen sets of executions, comprised of the following possible parameters (as summarized in Table 6):

Hashset of strings or single-value example application : we wanted to compare the performance impact of applications that have their state growing indefinitely, to applications that do not have a growing state.

CRC32 or SHA-2 for generating hashes : we wanted to compare the performance difference of each checksum algorithm;

Complete state serialization or state information implemented by application : we wanted to measure the performance impact of serializing all of the application state;

Message length impact: we nullified the hash included in the middleware protocol messages to measure the message length impact on performance.

All tests were executed during five minutes with one thousand requests per second, per instance, running three instances and each set was run fifty times. Logs were set to

	HS_CD	HS_CN	HS_CN_NM	HS_SD	HS_SN	HS_SN_NM	HS_OLD
OP/s	84.6834	2171.4857	2103.5544	76.1919	1835.8151	1988.9152	2723.1592
Deviation	0.8351	57.2670	50.7148	11.0445	57.1226	44.0784	50.4265
Deviation %	0.9861	2.6372	2.4109	14.4956	3.1116	2.2162	1.8518

Table 7: Performance of the hashset of strings example application

	SV_CD	SV_CN	SV_CN_NM	SV_SD	SV_SN	SV_SN_NM	SV_OLD
OP/s	2376.7394	2138.6497	2239.1337	1925.2754	1869.3208	2027.4538	2848.3247
Deviation	47.6534	42.8636	64.2503	44.9071	55.7483	47.7491	41.5389
Deviation %	2.0050	2.0042	2.8694	2.3325	2.9823	2.3551	1.4584

Table 8: Performance of the single-value example application

"ERROR" level so performance was not disrupted by logging.

3.4 Performance results and discussion

We measured average and standard deviation metrics for each test, based on their fifty runs. Please refer to Tables 7 and 8 for the results using the hashset of strings and single-value application examples, respectively. The deviation numbers show that, except for "HS_SD" test, there was no major performance discrepancies between the fifty runs in each test.

It is possible to see in Figure 14 the average performance of both applications during the fifty runs. The impact of serializing the complete application state for each example application is clearly visible and proves that the complete serialization is not suitable for applications that have their state growing indefinitely. The comparison between all variations, along with the statistical significance of these comparisons, can be found in Table 9. An independent-samples t-test (assuming significance for $p < 0.001$) was used to measure the differences, and it shows that there was a significant difference between all the compared tests, except for the pairs (SV_CN, HS_CN) and (SV_SN, HS_SN), meaning that both example applications have very similar performance when their state is not completely serialized.

Based on the performance impact of each comparison, we make the following observations:

CRC32 or SHA-2 for generating hashes: CRC32 provides some advantages over SHA-2 by having a smaller increase in message length, along with a smaller CPU performance impact and being good enough for capturing random bit flips. SHA-2 in the other hand provides complete bit flip coverage that may be necessary for ultra dependable systems (PAULITSCH et al., 2005);

Hashset of strings application or single value application: applications that have

Description	Variation 1	Variation 2	Impact	$t_{(50)}$	p-value
CRC vs SHA-2 in single-value application	SV_CN	SV_SN	12.59%	27.082	<0.0001
CRC vs SHA-2 in hashset application	HS_CN	HS_SN	15.46%	29.344	<0.0001
CRC vs SHA-2 without message overhead in single-value application	SV_CN_NM	SV_SN_NM	9.45%	18.698	<0.0001
CRC vs SHA-2 without message overhead in hashset application	HS_CN_NM	HS_SN_NM	5.45%	12.064	<0.0001
CRC Message overhead in single-value application	SV_CN	SV_CN_NM	-4.70%	-9.199	<0.0001
SHA-2 Message overhead in single-value application	SV_SN	SV_SN_NM	-8.46%	-15.234	<0.0001
CRC Message overhead in hashset application	HS_CN	HS_CN_NM	3.13%	6.279	<0.0001
SHA-2 Message overhead in hashset application	HS_SN	HS_SN_NM	-8.34%	-15.004	<0.0001
CRC State serialization in single-value application	SV_CN	SV_CD	-11.13%	-26.267	<0.0001
SHA-2 State serialization in single-value application	SV_SN	SV_SD	-2.99%	-5.527	<0.0001
CRC State serialization in hashset application	HS_CN	HS_CD	96.10%	257.642	<0.0001
SHA-2 State serialization in hashset application	HS_SN	HS_SD	95.85%	213.859	<0.0001
Original vs CRC hardening in single-value application	SV_OLD	SV_CN	24.91%	84.072	<0.0001
Original vs SHA-2 hardening in single-value application	SV_OLD	SV_SN	34.37%	99.574	<0.0001
Original vs CRC hardening in hashset application	HS_OLD	HS_CN	20.26%	51.123	<0.0001
Original vs SHA-2 hardening in hashset application	HS_OLD	HS_SN	32.59%	77.519	<0.0001
CRC single-value vs hashset application without state serialization	SV_CN	HS_CN	-1.54%	-3.246	0.0018
SHA-2 single-value vs hashset application without state serialization	SV_SN	HS_SN	1.79%	2.968	0.0038
CRC single-value vs hashset application with state serialization	SV_CD	HS_CD	96.44%	340.056	<0.0001
SHA-2 single-value vs hashset application with state serialization	SV_SD	HS_SD	96.04%	282.731	<0.0001

Table 9: Summary of impacts of Variation 1 against Variation 2

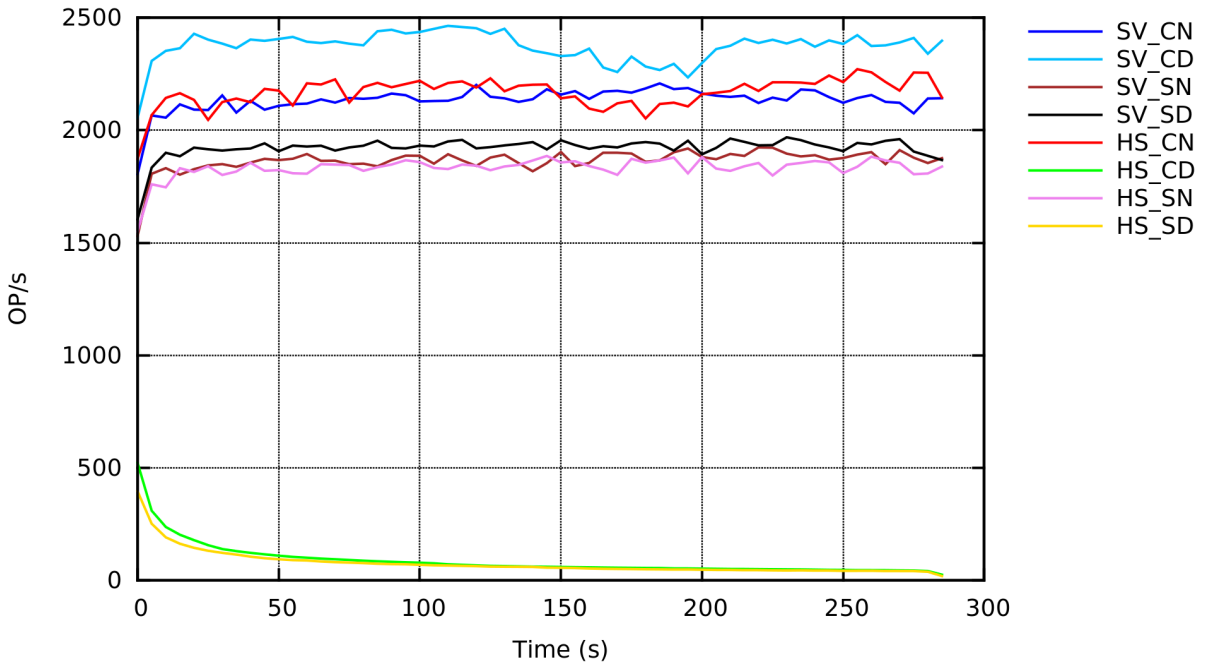

Figure 14: State serialization impact in both example applications

their state growing indefinitely suffer from a huge performance impact if their state is completely serialized, while also suffering from much increased memory usage. Applications that have a smaller, fixed-size state can benefit from complete state serialization to achieve better coverage with minimal performance impact;

Complete state serialization or state information implemented by application: complete state serialization allows perfect application state coverage, while taking a performance impact proportional to the size of application state. Whenever the application can implement a method to return its state information, the performance gain is inversely proportional to the size of the application state;

Message length impact: the hash included in middleware protocol messages are responsible for minimal performance impact.

Overall, we believe that our implementation techniques, along with the customization level we provided by being able to select the hashing algorithm and the application-implemented state information method, shows performance that is evidence of the feasibility of our proposal. The state information method implemented in our hashset of strings example application, combined with the semantic checks implementation, was a very cost-effective solution. This approach was much faster than the complete serialization of state approach to generate the state information, despite not having to implement the semantic check in this case. The most obvious drawback was that any corruption in the elements in memory are not detectable. The implementation of different techniques that can

provide additional coverage is delegated to the application. We considered implementing the arithmetic codes solution (BEHRENS; WEIGERT; FETZER, 2013) as a third, more efficient alternative to complete state serialization, but we felt this was out of the main scope of this dissertation. For any application that is not ultra dependable or safe-critical, we recommend the use of CRC32, although given the performance difference of CRC32 and SHA-2, it is not a very big performance compromise if SHA-2 is used instead.

Conclusion

Among the fault models used to build distributed systems, crash-recovery and arbitrary stand out for benign and arbitrary fault classes, respectively. There is a big difference in types of faults tolerated and also in resource requirements for each of those fault models, where arbitrary has not been the preferred model. It is possible to propose a set of error detection techniques that allows benign crash-recovery algorithms to be hardened towards the same coverage as arbitrary algorithms while excluding malicious faults. By implementing and experimenting with those techniques on a Paxos-based library, we hardened our fault model, successfully tolerating non-malicious arbitrary faults and achieving a crash-stop non-malicious arbitrary fault model by aborting the replica execution once an error is detected. Our work currently does not recover from such faults, but at this point, we consider the crash-stop non-malicious arbitrary fault model to be more resilient and more practical than arbitrary, also requiring less effort on developers to create a fault tolerant application for this fault model using such middleware.

3.5 Opportunity for future work

Among items that we could not accomplish within the scope of this research we can list:

Fault model: In order to achieve the non-malicious arbitrary crash-recovery fault model as we initially intended, all crash occurrences must be recoverable. As discussed in a previous section, a state transfer mechanism could be the approach taken for many of our implemented detections that are mapped to crash-stop, so they can be mapped to crash-recovery. Such mechanism is not present in our middleware of choice. Some study on requirements to implement it indicate that it requires a state information structure, which we implemented in this research. Using the state count and checksum, replicas can validate their state before transferring, and know exactly to which point in time they need to have their state transferred to.

Coverage: Unless all the state is being serialized, the coverage on the state may not be guaranteed. Other approaches investigated, such as the arithmetic codes solution (BEHRENS; WEIGERT; FETZER, 2013), may provide better overall coverage in the Paxos algorithm by detecting local errors, in contrast to our distributed validation mechanism which detects divergences between replicas.

Performance: It is clear that the biggest discrepancy lies in the choice of the state information approach. We believe that using a third approach such as arithmetic

codes (BEHRENS; WEIGERT; FETZER, 2013), as pointed in our performance results discussion, could offer an alternative that fits the application type better than both solutions offered in our research. Any other approach used to improve the fault model or coverage may degrade performance and end up being disadvantageous compared to what our research already offers.

The improvements listed above, if implemented, would make our hardened Paxos-based middleware even more resilient against non-malicious arbitrary faults, and more practical by being fully recoverable.

Publications

Title: Hardened Paxos through Consistency Validation

Published in: 2015 Brazilian Symposium on Computing Systems Engineering (SBESC)

DOI: http://dx.doi.org/10.1109/SBESC.2015.10

Bibliography

BEHRENS, D.; WEIGERT, S.; FETZER, C. Automatically tolerating arbitrary faults in non-malicious settings. In: *Dependable Computing (LADC), 2013 Sixth Latin-American Symposium on*. [S.l.: s.n.], 2013. p. 114–123. Citado 9 vezes nas páginas 14, 15, 24, 26, 28, 29, 56, 57, and 58.

BHATOTIA, P. et al. Reliable data-center scale computations. In: *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*. New York, NY, USA: ACM, 2010. (LADIS '10), p. 1–6. ISBN 978-1-4503-0406-1. Disponível em: <http://doi.acm.org/10.1145/1859184.1859186>. Citado 7 vezes nas páginas 14, 15, 23, 24, 26, 27, and 29.

BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006. (OSDI '06), p. 335–350. ISBN 1-931971-47-1. Disponível em: <http://dl.acm.org/citation.cfm?id=1298455.1298487>. Citado na página 21.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. *Introduction to reliable and secure distributed programming*. [S.l.]: Springer, 2011. Citado 2 vezes nas páginas 13 and 17.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 20, n. 4, p. 398–461, nov. 2002. ISSN 0734-2071. Disponível em: <http://doi.acm.org/10.1145/571637.571640>. Citado 3 vezes nas páginas 14, 22, and 23.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: An engineering perspective. In: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2007. (PODC '07), p. 398–407. ISBN 978-1-59593-616-5. Disponível em: <http://doi.acm.org/10.1145/1281100.1281103>. Citado 4 vezes nas páginas 15, 26, 27, and 28.

CLARKE, D. et al. Incremental multiset hash functions and their application to memory integrity checking. In: LAIH, C.-S. (Ed.). *Advances in Cryptology - ASIACRYPT 2003*. Springer Berlin Heidelberg, 2003, (Lecture Notes in Computer Science, v. 2894). p. 188–207. ISBN 978-3-540-20592-0. Disponível em: <http://dx.doi.org/10.1007/978-3-540-40061-5_12>. Citado na página 26.

CORREIA, M. et al. Practical hardening of crash-tolerant systems. In: *USENIX Annual Technical Conference*. [S.l.: s.n.], 2012. p. 453–466. Citado 8 vezes nas páginas 13, 14, 15, 23, 24, 26, 27, and 29.

GUERRAOUI, R.; SCHIPER, A. Fault-tolerance by replication in distributed systems. In: STROHMEIER, A. (Ed.). *Reliable Software Technologies — Ada-Europe '96*. Springer Berlin Heidelberg, 1996, (Lecture Notes in Computer Science, v. 1088). p. 38–57. ISBN 978-3-540-61317-6. Disponível em: <http://dx.doi.org/10.1007/BFb0013477>. Citado 2 vezes nas páginas 13 and 17.

LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 16, n. 2, p. 133–169, maio 1998. ISSN 0734-2071. Disponível em: <http://doi.acm.org/10.1145/279227.279229>. Citado 3 vezes nas páginas 14, 18, and 29.

LAMPORT, L. Paxos made simple. *ACM Sigact News*, v. 32, n. 4, p. 18–25, 2001. Citado 3 vezes nas páginas 18, 20, and 29.

LAMPORT, L. Fast paxos. *Distributed Computing*, v. 19, n. 2, p. 79–103, 2006. ISSN 1432-0452. Disponível em: <http://dx.doi.org/10.1007/s00446-006-0005-x>. Citado na página 29.

LAMPORT, L. Byzantizing paxos by refinement. In: *Proceedings of the 25th International Conference on Distributed Computing*. Berlin, Heidelberg: Springer-Verlag, 2011. (DISC'11), p. 211–224. ISBN 978-3-642-24099-7. Disponível em: <http://dl.acm.org/citation.cfm?id=2075029.2075058>. Citado 2 vezes nas páginas 14 and 23.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 4, n. 3, p. 382–401, jul. 1982. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/357172.357176>. Citado 2 vezes nas páginas 22 and 24.

LAMPSON, B. W. How to build a highly available system using consensus. In: *Proceedings of the 10th International Workshop on Distributed Algorithms*. London, UK, UK: Springer-Verlag, 1996. (WDAG '96), p. 1–17. ISBN 3-540-61769-8. Disponível em: <http://dl.acm.org/citation.cfm?id=645953.675640>. Citado 2 vezes nas páginas 14 and 21.

LEFF, A.; RAYFIELD, J. T. Web-application development using the model/view/controller design pattern. In: *Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International*. [S.l.: s.n.], 2001. p. 118–127. Citado na página 30.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). *SHA-2 Standard*. 2002. Secure Hash Standard. FIPS PUB 180-2, <www.itl.nist.gov/fipspuhs/fip180-2.htm>. Citado na página 40.

OKI, B. M.; LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 1988. (PODC '88), p. 8–17. ISBN 0-89791-277-2. Disponível em: <http://doi.acm.org/10.1145/62546.62549>. Citado 2 vezes nas páginas 14 and 20.

PAULITSCH, M. et al. Coverage and the use of cyclic redundancy codes in ultra-dependable systems. In: IEEE. *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. [S.l.], 2005. p. 346–355. Citado na página 53.

PETERSON, W. W.; BROWN, D. T. Cyclic codes for error detection. *Proceedings of the IRE*, v. 49, n. 1, p. 228–235, 1961. Citado na página 40.

RENESSE, R. van; SCHIPER, N.; SCHNEIDER, F. Vive la différence: Paxos vs. viewstamped replication vs. zab. *Dependable and Secure Computing, IEEE Transactions on*, PP, n. 99, p. 1–1, 2014. ISSN 1545-5971. Citado 2 vezes nas páginas 14 and 21.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 22, n. 4, p. 299–319, dez. 1990. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/98163.98167>. Citado 4 vezes nas páginas 14, 18, 22, and 23.

SCHWARZ, T. et al. Disk scrubbing in large archival storage systems. In: *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on.* [S.l.: s.n.], 2004. p. 409–418. ISSN 1526-7539. Citado na página 26.

TORVALDS, L. *Git.* 2005. <https://git-scm.com/>. Accessed: 26-May-2016. Citado na página 33.

VIEIRA, G. M. D.; BUZATO, L. E. Treplica: ubiquitous replication. In: *SBRC'08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems.* [S.l.: s.n.], 2008. Citado 2 vezes nas páginas 15 and 29.

VIEIRA, G. M. D.; BUZATO, L. E. Implementation of an object-oriented specification for active replication using consensus. In: . [S.l.]: Technical Report IC-10-26, Institute of Computing, University of Campinas, 2010. Citado 2 vezes nas páginas 15 and 29.