University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Master of Science in Computer Science	Spring semester, 2013 Open
Writer: Tormod Erevik Lea	..... (Writer's signature)
Faculty supervisor: Hein Meling	
Thesis title: Implementation and Experimental Evaluation of Live Replacement and Reconfiguration	
Credits (ECTS): 30	
Key words: Distributed Systems Replicated State Machines Failure Handling	Pages: 94 Enclosure: CD Stavanger, 1 July 2013

Front page for master thesis
Faculty of Science and Technology
Decision made by the Dean October 30th 2009

FAILURE HANDLING

for

PAXOS STATE MACHINES

Implementation and Experimental Evaluation of

Live Replacement and Reconfiguration

Tormod Erevik Lea

June 2013

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology
University of Stavanger

Abstract

State machine replication is a common applied technique for building fault-tolerant services. The technique uses a collection of replicas to mask failures. All replicas are provided the same sequence of operations (requests), resulting in that they end up in a consistent state. A consensus protocol such as Paxos is normally used to the order request issued by multiple clients to a Replicated State Machine (RSM). RSMs achieve high availability by replicating state across several machines. Such an approach enables access to state stored in system even in the presence of failures. An RSM is prohibited from processing new requests if more than half of its replicas fail. It is therefore important that replica failures are handled and repaired as soon as possible for keeping the availability and fault-tolerance of the RSM high. This thesis presents an description and implementation of two existing methods for immediate failure handling for Paxos-based RSMs: Live Replacement and Reconfiguration. Both failure handling methods have been implemented as part of the Goxos framework. An experimental evaluation and comparison of the two methods is also presented.

Acknowledgments

I would like to thank my supervisor, Associate Professor Hein Meling, for guidance and invaluable feedback throughout my work on this thesis.

I would also like to thank Leander Jehl, who has been very helpful in explaining the Live Replacement protocol, in addition to providing much valuable feedback on the thesis as a whole.

Finally, I would like to thank my family for all their support during my Master's degree.

Contents

1 Introduction

All large web sites and web applications today usually consist of several distributed subsystems. Such subsystems can for example be web servers, databases, application servers and load balancers. All these systems must normally coordinate and exchange information for keeping the web site or application functioning properly. A web server may for example need to know what database is acting as primary if a primary/backup model is used for replication. Many complex systems use a separate configuration database for providing individual subsystems with such information. This database needs to be highly available so that the subsystems always receive a response to any request they send. For being highly available the configuration database need to tolerate failures. The configuration database must often also provide consistency, meaning that the subsystems see the same data at the same time.

A special configuration database of the type described above is often implemented as a Replicated State Machine (RSM) by using the state machine approach [1]. State machine replication is a common applied technique for building fault-tolerant services. The technique uses a collection of replicas to mask failures. All replicas are provided the same sequence of operations (requests), resulting in that they end up in a consistent state. A consensus protocol such as Paxos [2] is normally used to the order request issued by multiple clients to an RSM. RSMs achieve high availability by replicating state across several machines. Such an approach enables access to state stored in system even in the presence of failures. An RSM is prohibited from processing new requests if more than half of its replicas fail. It is therefore important that replica failures are handled and repaired as soon as possible for keeping the

availability and fault-tolerance of an RSM high. This thesis presents an description and implementation of two existing methods for immediate failure handling for Paxos-based RSMs: Live Replacement and Reconfiguration.

1.1 Contributions and Outline

This thesis contributes a proof-of-concept implementation of *Live Replacement* [3], a new failure handling method for Paxos State Machines [2]. Live Replacement is a technique stated to be specifically targeted at immediate failure handling. The failure handling technique has been implemented as part of the Goxos framework [4]. Goxos is a Paxos-based RSMs framework that was created at the University of Stavanger in 2012. The thesis provides a description of both the design and implementation work done for incorporating Live Replacement as a part of the Goxos framework. A special focus is put on implementation specific details not addressed in the theoretical description of Live Replacement.

One of the main motivations for this work is to evaluate Live Replacement against other applicable methods for immediate failure handling. An already well known failure handling method, *Reconfiguration* [5], is for this reason also implemented as part of the work this thesis. The implementation of Reconfiguration as part of the Goxos framework is thoroughly described in this work.

Both presented failure handling methods are experimentally evaluated and compared through a set of experiments. The obtained experimental results are presented and analyzed as a part of this thesis.

The remainder of this thesis is organized as follows:

Chapter 2 describes the assumed system model for this work. The chapter also defines a set of metrics for immediate failure handling.

Chapter 3 introduces relevant background material for this thesis. The main focus is on the Paxos protocol [2] and how it is used to build Replicated State Machines [1].

Chapter 4 presents Goxos [4], a Paxos-based Replicated State Machine framework, on which the implementation work for this thesis is based.

Chapter 5 provide a theoretical description of three failure handling methods: Recovery, Reconfiguration [5] and Live Replacement [3].

Chapter 6 describes the design and implementation of two failure handling methods, Live Replacement and Reconfiguration, implemented as part of the Goxos framework.

Chapter 7 presents a set of experimental results from a single failure scenario, using both Live Replacement and Reconfiguration. The chapter also provide an evaluation of the two methods in light of the observed experimental results.

Chapter 8 concludes and presents some suggestions for further work.

2 Model and Definitions

This chapter presents the system model assumed for this thesis, as well as a set of metrics for immediate failure handling methods.

2.1 System Model

The context of this thesis is a distributed system which features a set of nodes communicating by sending messages in a communication network. A node can either host a replicated server process, denoted *replica*, or a client process.

The system is defined to be *partially synchronous*. This means the system normally is assumed to be *synchronous*, but that there may be periods where the system is *asynchronous*. A asynchronous system makes no timing assumptions about process or links, while a synchronous system has a known upper bound for both processing and message transmission delays. A partially synchronous system can be said to capture the assumption that a system may not always be synchronous, and that the resulting asynchronous periods has no bound on duration. Such an assumption maps well to practical systems, where for example both malfunctioning hardware and unexpected software behavior can cause periods of asynchrony.

The system's communication network is assumed to be reliable. Messages can take an arbitrarily long time to arrive, can be duplicated and can be lost, but not corrupted. The network may partition into several disjoint components, and components may later be consolidated. This assumption is made despite the usage of the reliable Transmission Control Protocol (TCP) for all network communication

in the implementation work done for this thesis. The consensus protocol and failure handling methods relevant for this thesis do not require reliable communication because they use protocol specific mechanisms to handle possible message loss.

The fault model for replicas is assumed to be crash-stop [6, p. 24]. Replicas can in this model fail by crashing and crashes are permanent. Abstractly, a process in this model is said to be *correct* if it never crashes and executes an infinite number of steps. A process that crashes at some time during the execution is defined as *faulty*. As a result of the defined fault model it is noted that replicas do not fail in an arbitrary manner. This means that they do not deviate in any way from the algorithm assigned to them (exhibit Byzantine behavior). As elaborated later in Section 2.2, this thesis focuses on immediate failure handling and does not consider Recovery. Due to the defined permanent failure model, it is finally mentioned that for the scenarios considered in this thesis, replicas do not write to stable storage during operation. It is also assumed that the system has additional resources available for starting or obtaining new replica instances.

The replicated service is assumed to be implemented as a deterministic state machine. They together form a Replicated State Machine (RSM), which provides a stateful deterministic service to clients. The RSM is defined to use the Paxos consensus protocol [2] for ordering updates to the state machines. The replicated service consists of n replicas. Using Paxos, such as service need $n = 2f + 1$ replicas to tolerate f failures. Paxos guarantees safety, but can not guarantee liveness due to possible periods of asynchrony. The concept of Replicated State Machines, Consensus and the Paxos Protocol are further described in Chapter 3. Clients act by sending requests as messages to a cluster of replicas. Each request can be uniquely identified from a monotonically increasing sequence number. It is assumed that clients can only have one request outstanding at a time. Thus client c will only issue a new request with sequence number $i_c + 1$ when it has received a response for request number i_c. Clients are also assumed not to send any malformed requests or exhibit any other form of Byzantine behavior.

2.2 Metrics for Immediate Failure Handling

As motivated in Chapter 1, this thesis focus on methods for immediate failure handling. A Paxos State Machine with $n = 2f + 1$ replicas can tolerate up to f failures. Any real-world system clearly do not respect this bound. Scenarios involving more than f concurrent failures will make it impossible for an RSM to make progress until the completion of a possibly time-costly human-operated repair procedure. Such critical scenarios also significantly reduces the system's availability.

Durations with pending failure handling is clearly a critical and vulnerable period for any real-world system implemented as an RSM. A system could experience

additional successive failures during this period, quickly exceeding to bound set by f. In this context we define a *window of vulnerability*, similar to how it is defined in [7]. This work will refer to this window as any period where a system is not operating at its initial level of fault-tolerance.

Replica failures should be handled fast and efficiently using an appropriate failure handling method to avoid critical scenarios such as the ones mentioned above. In addition to being fast, such methods should also have minimal impact on state machine progress. Reduced state machine progress is directly visible to clients through increased request latencies during periods of failure handling.

A goal of this thesis is an experimental evaluation of two applicable methods for such immediate failure handling. For comparing and evaluating their performance a set of metrics are needed. This text uses the metrics defined in [3] as a basis. Two main metrics are defined below which an efficient failure handling method should minimize:

- **Delay** is defined as the time from failure detection until a new replica is participating in Paxos. For the experimental evaluation this metric is naturally a time measurement. It is noted that failure detection time is not included in this metric, since detection time is a tunable parameter. The experimental results will identify and explicitly state what part of the failure handling procedure is detection time.
- **Disruption** is defined as the additional latency state machine requests experience during failure handling. For the experimental evaluation this metric will be measured from clients points of view. The experienced request latency observed by clients will be recorded.

In addition, two other metrics are presented below, addressing potential causes for increased Delay and Disruption. They are not directly applicable to experimental evaluation, but are useful for the theoretical descriptions given in Chapter 5.

- **Potential Delay** is the set concurrent protocol steps required for a new replica to start.
- **Potenital Disruption** is the set of protocol steps required to avoid disruption.

3 Background

In this chapter the relevant background material related to this thesis is presented. The section presents a set of fundamental concepts related to State Machine Replication, Consensus and the Paxos Protocol. Finally, an introduction to The Go Programming Language is given. It is noted that some of the background material presented here uses [4] as a basis. The source is a collaborative report, co-written with Stephen M. Jothen as part of a preliminary project in the Fall 2012 semester.

3.1 State Machine Replication

A well known approach for constructing fault tolerant distributed services is the State Machine Approach [1]. This technique uses a collection of replicas to mask failures. All replicas are provided the same sequence of operations (requests), resulting in that they end up in a consistent state and provide the same sequence of outputs (responses). A challenge for RSMs arises when there exists multiple clients. Concurrent requests from different clients may arrive in different order at the replicas, forcing them to make differing transitions. This can naturally lead to replicas producing different outputs and ending up in an inconsistent state. A consensus protocol can be used to solve this challenge, and ensure that an RSM behaves identically to a single remote state machine (that never crashes).

3.2 Consensus

Consensus in one of the most fundamental problems in distributed computing, and the consensus abstraction underlies many systems and protocols. The reason for its importance is related to algorithms operating in a model that allows failure. Algorithms that provide several processes to maintain a common state or to decide on a future action all rely on solving a consensus problem. The abstraction is used by processes to reach agreement on a single value from a set of proposed candidate values. Consensus has been shown to be impossible to solve in a fully asynchronous system, even if only one process fails and it can only do so by crashing [8]. However, this result does not state that consensus never can be reached. It states that no algorithm can reach consensus in bounded time when applying the assumptions for the defined system model. Reformulated, it is proved that any fault-tolerant algorithm for solving consensus has runs that never terminate. Such runs are still very unlikely to happen.

Distributed algorithms implementing a distributed programming abstraction (such as consensus), needs to meet a certain set of properties that hold in all possible executions of the algorithm. This set of properties are usually divided into two separate classes: *safety* and *liveness*. Summarized, it can be said that safety means that the algorithm not should do anything wrong, while liveness ensures that eventually something good happens. For consensus abstractions, the set of safety and liveness properties are traditionally stated as [6]:

CL1 Termination: Every correct process eventually decides some value.

CS1 Validity: If a process decides v, then v was proposed by some process.

CS2 Integrity: No process decides twice.

CS3 Agreement: No two correct processes decide differently.

CL1 is the only property for consensus related to liveness. It states that algorithm should eventually terminate by having every process decide some value. CS1-3 all define safety properties. CS3 defines the underlying main goal of consensus, namely that every two correct processes actually decides on the same value.

3.3 The Paxos Protocol

Paxos is a family of consensus protocols that can be used to achieve ordering of requests issued to a replicated state machine. Paxos was created by Leslie Lamport and its initial description [9] was published in 1998 after almost 10 years after its invention. In the original paper, Lamport describes the Paxos algorithm in terms

of the fictional parliamentary system on the Greek island of Paxos. Many people found this way of defining the algorithm to be confusing, which led to Lamport creating a simpler explanation of Paxos published three years later [2].

In [2], the Paxos algorithm is described through the concurrent interaction between three independent types of processes, or agent roles:

- **Proposers** can propose values to agree on.
- **Acceptors** accept a value among those proposed.
- **Learners** learn chosen values.

A process can (and in practice normally does) assume all three agent roles. The original Paxos protocol (often referred to as *single-decree* Paxos) can be used to agree and decide on a *single* value out of values initially proposed. Single-decree Paxos is composed of two phases described below, and visualized in Figure 3.1. Phase 1 is only necessary when a new leader takes over. Phase 2 can be executed to decide on a given client request.

- **Phase 1:** *Determine if it is safe to propose any value or obtain a value already voted for.*

  a) A single proposer among a group of processes is elected to be the leader. The leader broadcasts a PREPARE message with round number $crnd$.[1] Each proposer maintains a set of of predetermined increasing round numbers disjoint from other proposers. This ensures that no two proposers can use the same round number. Acceptors only cooperate with the proposer with the highest round number.

  b) When an acceptor receives a PREPARE message with round number $crnd$, and $crnd$ is greater than any previous PREPARE message received, it stops cooperating with proposers using lower round numbers, and responds with a PROMISE message containing the previous highest round number ($vrnd$) and the associated value ($vval$). This is the value that it voted for in its last sent LEARN message (Phase 2b).

  c) If the proposer receives a majority of PROMISE messages for round number n, it is free either propose the safe reported value, or any value if no acceptor reports a value.

[1] The term *ballot* is often also used in the literature for referring to this number. This text uses *round number*.

- **Phase 2:** *Attempt to get a safe value accepted.*

  a) If the response from the majority of acceptors contained a bounded value, the proposer broadcasts an ACCEPT message containing *crnd* and the value (*cval*) associated with the maximum round number of the received PROMISE messages. If the response from the acceptors left the Proposer unconstrained to chose any value, it may wait until it receives a client request and use this value for the ACCEPT message.

  b) If an acceptor receives an ACCEPT message with a round number (*crnd*) greater or equal to the one it has already seen (*vrnd*), it broadcasts a LEARN message to the learners containing the associated round and value.

  c) When learners receive LEARN messages from a majority of acceptors for this round number, the command is said to be decided and can be executed. A majority is referred to as a *quorum*, and is the set of at least half of the processes or $\lfloor n/2 \rfloor + 1$, in a crash-fault tolerant system.

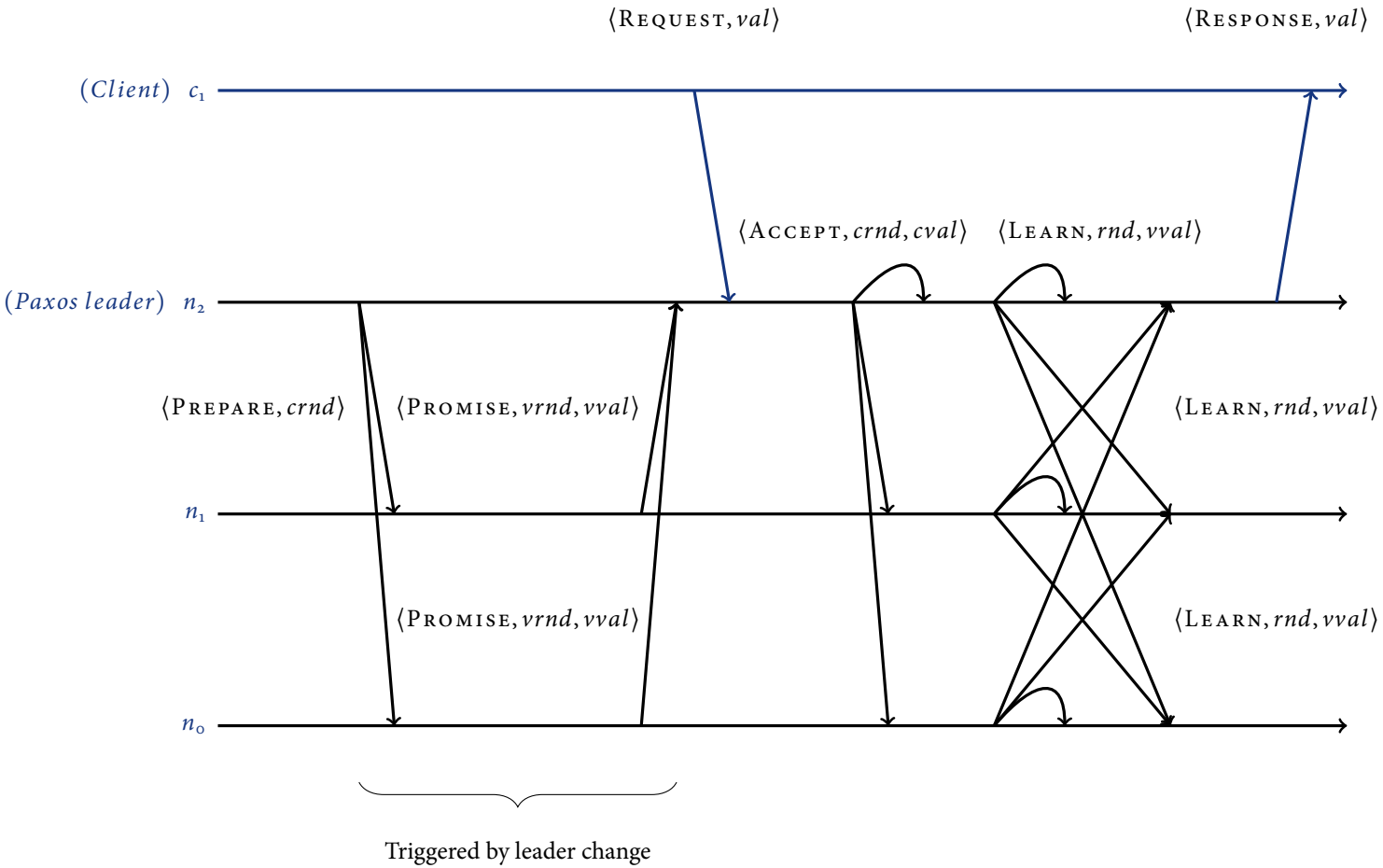

Figure 3.1: Single-decree Paxos

3.4 The Paxos State Machine

Single-decree Paxos is, as discussed above, used to agree on a single value. An RSM typically needs to order a continuous stream of incoming client requests. To create a Paxos State Machine, the Paxos protocol is expanded to a variant called *multi-decree* Paxos. This extension starts one instance of the basic single-decree variant for each request received. During operation the protocol assigns the decided values from these instances to an ordered sequential range of *slots* according to a consensus instance number.

Multi-decree Paxos can be optimized by letting the leader execute phase 1 for an arbitrary number of consensus instances [2, p. 9]. As a result, during periods of synchrony, this variation reduces the number of communication steps needed, from four to two per consensus instance. When this optimization is used, round numbers span horizontally over all concurrent instances.[2]

Multi-decree Paxos performance can in certain scenarios also benefit from other optimizations, among them a technique called pipelining. Pipelining allows the protocol to execute a certain number of Paxos instances concurrently. This can be done as long as execution of decided commands are done in correct order. The number of allowed concurrent instances is limited by the pipelining parameter, traditionally denoted as α. Applying pipelining can improve utilization of resources but requires careful environment-specific tuning of the pipelining parameter.

Another method for achieving additional throughput for Paxos systems is a technique called *batching*. Batching combines multiple received commands into a single Paxos instance. As a consequence, batching benefits from spreading the fixed per-instance cost over several request. It is often stated to be an optimization that can provide largest gain in terms of performance, but need to consider the trade-off related to increased client latencies when waiting for a batch to fill up or for the batching interval to timeout. A thorough investigation of both batching and pipelining can be found in [10].

The combination of multi-decree Paxos and pipelining introduces changes for how safe values for proposal should be obtained (Phase 1). When using $\alpha > 1$, a leader may fail before finishing broadcasting accepts for several slots. As a consequence there may be several undecided requests (globally limited by α). For handling this a proposer that considers itself to be the Paxos leader (using an external leader detection mechanism), issues a ⟨PREPARE⟩ using a round number higher than any previously seen. The message also contains a slot identifier, i, denoting the highest slot in its continuous range of decided instances. Acceptors responds to this message with a ⟨PROMISE⟩ containing the values for each slot higher than

[2] Round numbers are sometimes in combination with multi-decree Paxos referred to as *views*. A system is said to transition to a new view on leader change.

i in which it has accepted a value. The proposer can then safely propose values for slots greater than i. This can be optimized further if the acceptor and learner are co-located. The $\langle$PROMISE$\rangle$ can then contain either a decided value or the highest value accepted for each slot greater than i. This avoids already decided slots to become re-decided.

Figure 3.2 visualize the utilization of multi-decree Paxos. The scenario in question uses $\alpha = 3$ and assume that Phase 1 of the protocol has already been executed. In the figure four different clients each send a request to the Paxos leader. The Paxos leader starts a new Paxos instance for the three first requests it receives. These instances may be *decided* concurrently, but must be *executed* in order. The next request received from c_4 can, due to the restriction set by the pipelining parameter, not be proposed by the leader right away. When the Paxos protocol decides on the first request sent by c_1, it issues it a response to the client. It is afterwards able to propose the request from c_4, since there are now only two concurrent instances being decided. This can be viewed as a sliding window of requests.

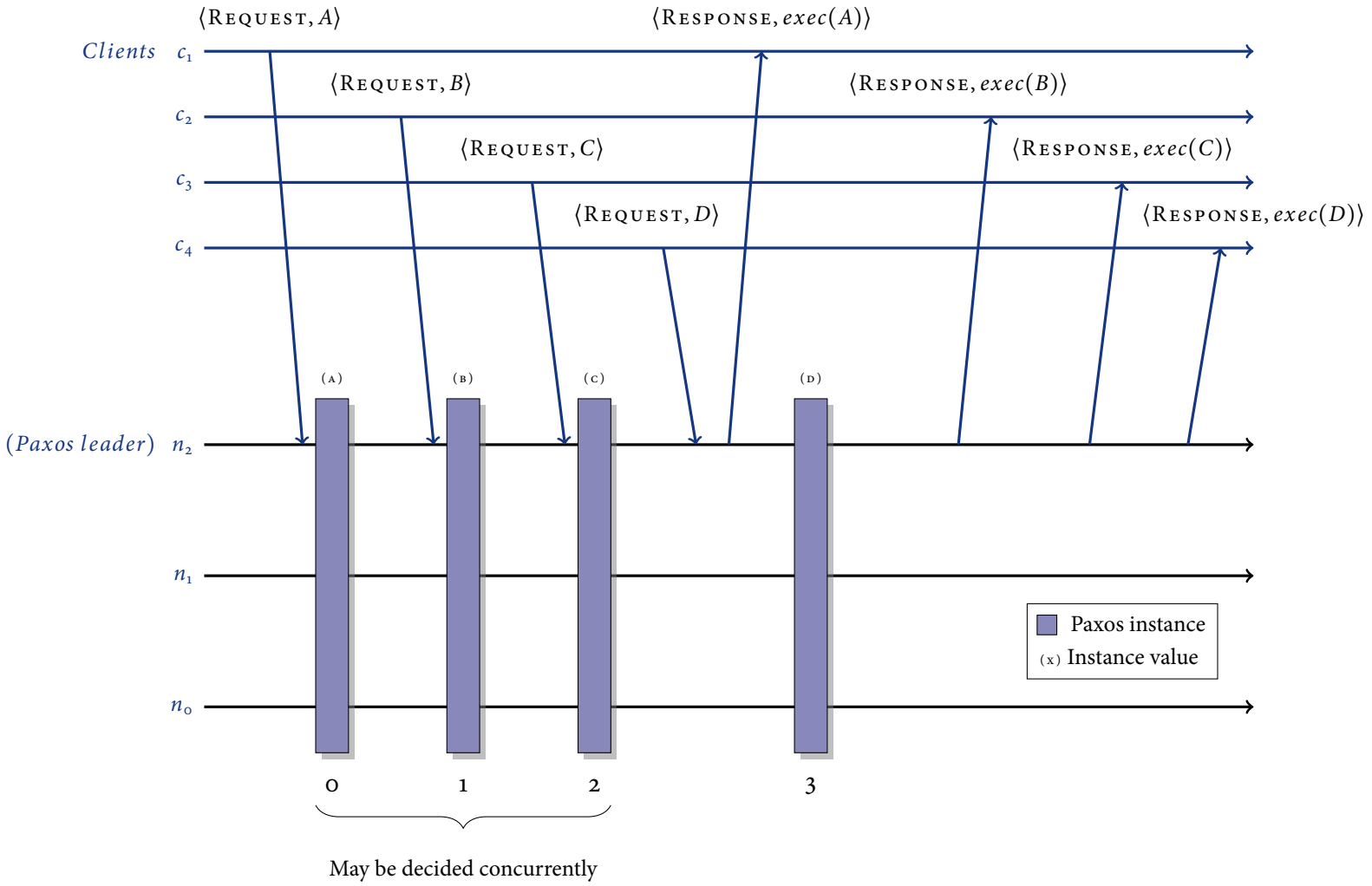

Figure 3.2: Multi-decree Paxos ($\alpha = 3$)

3.5 The Go Programming Language

The Goxos RSM framework, which will be later introduced in Chapter 4, forms the foundation for the implemented failure handling techniques discussed in this thesis. The framework is implemented in the Go programming language [11], and its developed extensions makes heavy use of Go's concurrency model. This model is based on Communicating Sequential Processes (CSP) [12] developed by C. A. R. Hoare in 1978. Horare introduced the concept of using channels for interprocess communication and synchronization. The CSP related language constructs are not found in most mainstream programming languages. For this reason a short introduction to Go and its particularities is given here.

Go was initially designed and developed by Google in 2007 and was later released as an open-source project in 2009. Go is often referred to as a system programming language. It achieves good performance as a result of being a statically typed and compiled language. As opposed to related languages such as C and C++, Go also features a garbage collector and is memory safe.

Todays ubiquitous presence of multiple cores (and even multiple processors) in both traditional computers and mobile devices requires developers to implement concurrent programs in order to harness more of the computing power available. Traditionally, writing concurrent programs using mechanisms such as threading and locking has often proved to be both difficult and error-prone. Complex memory management and data races are often cited problems in this context.

Go's CSP approach to concurrency attempts to make implementing concurrent programs easier for the developer. The language authors advocates passing ownership of shared data over channels, and often refers to the mantra "Do not communicate by sharing memory; instead, share memory by communicating" [13].

Go provides two main high-level facilities for concurrent programming: goroutines and channels. A goroutine is defined as "a function executing concurrently with other goroutines in the same address space" [13]. Goroutines are spawned by prefixing a function or method call with the `go` statement. A channels is a "a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type" [14]. Go uses arrow notation to either specify a send statement (`chan<-`) or a receive operation (`<-chan`) on a channel. Channels can be unbuffered or buffered with a certain capacity to allow asynchronous communication. The `select` statement is often used in conjunction with channels. It can be compared to a traditional `switch` statement, but instead operates on channels where it chooses which of a set of possible communications will proceed.

The initial work on Goxos found Go's concurrency model to be an excellent match that allowed us to sidestep much of the complexity of threading and locking that are widespread in other languages [4]. Also, using Go enabled an implemen-

tation of Paxos very similar to how the algorithm is described in the original papers [2, 9]. The use of goroutines and channels for implementing the independent and concurrent Paxos agents proved to work very well.

4 Goxos

The implementation work done as a part of this thesis builds on Goxos [4], a Paxos-based RSM framework. The framework has been further extended from its initial form to include the failure handling capabilities discussed and analyzed in this thesis. Since the discussions presented in this work will refer to different parts of the Goxos framework extensively, an architectural overview of the framework is given together with a short description of the relevant submodules.

4.1 History

The work on Goxos was initiated by the Distributed Systems Group at the University of Stavanger in the autumn of 2012. The main motivation was to create a platform for implementing and testing various research subjects such as

- Paxos variations
- Failure handling for Paxos
- Fault tolerant publish/subscribe

The main foundation of Goxos was implemented by two master students taking the course Project in Computer Science. This development also served as preparatory work for our upcoming Master's theses. Developing an modular and extensible RSM framework from scratch is an extensive undertaking [15]. Hence, combined with

the limited time available for such an initial project, Goxos was in a rudimentary shape when the work for this thesis started. Although the main objective of this thesis were to implement the relevant failure handling methods, a great deal of work has also gone into adjusting, extending and maintaining the core Goxos framework. This work was necessary in order to fulfill the goals set for this thesis. To give an idea of the effort involved, some of the main tasks are given below. Architectural references mentioned will be explained in Section 4.3.

- **Client integration:** The initial Goxos framework did not contain any form of client handling. As a consequence, both a client handler and separate client library were developed.
- **Service integration:** No integration against the replicated service were available when the work for this thesis began. The interface for connecting a replica to the replicated service were defined and implemented in the initial work phase.
- **Applications:** The Goxos project were not bundled with any applications acting as replicated services. A few example applications were implemented to enable proper testing during development.

In addition to the points mentioned above, a development project of Goxos' size obviously continuously need refactoring, bug fixes and adjustments. Such tasks has also required a great deal of effort.

4.2 Overview

The Goxos RSM framework provides a consensus service to applications. Applications can use the framework to implement a consistent stateful service. The corresponding client library can be used for letting clients issue requests to the service. The framework can be configured to use different Paxos implementations available in Goxos, including MultiPaxos, FastPaxos and AggregatedPaxos [16]. Fault-tolerance is achieved by replication and maintained through utilization of the failure handling methods implemented for this thesis.

Consistent with the model stated in Section 2.1, a system using Goxos runs on a set of *nodes* communicating with each other using message passing. A node will in a typical deployment be an independent single machine communicating over the network.

4.3 Architecture

The Goxos project in its current form consists of three main components (and source code repositories):

- **goxos**: The main framework for building Paxos-based RSM.
- **goxosc**: A client library for implementing application clients used for issuing requests to a Goxos cluster.
- **goxosapps**: A collection of utilities and example applications utilizing the Goxos framework. The component contains a simple replicated key-value store, KVS, that is used in the experiments described in Chapter 7.

The discussion here will mainly focus on the Goxos framework, since it forms the basis for the implementation work for done for this thesis. The mentioning of Goxos throughout this thesis will refer to the main RSM framework unless otherwise stated.

The Goxos framework consists of a group of Go packages, each representing a separate part of the framework's functionally. Each package again contains one or more components that will be referred to as *modules*. A module in the context of Goxos can be described as an actor handling a specific responsibility. Each module runs in a separate goroutine, using an event loop to react to messages from other parts of the system. The event loops are constructed using Go's `select` keyword and channels for inter-module message passing.

Listing 4.1 shows a simple and representative example of such an event loop. The example is taken from the Learner module in the MULTIPAXOS package. The arrow notation shows operation on channels, and for this example more precisely the usage of the receive operator on a set of channels. The messages received by the module is passed on to the appropriate handling method. For example is every ⟨LEARN⟩ message received from the `learnChan` passed on to the `HandleLearn` method. The CSP-related usage of `select` and channels obviates the need for any locking, since the module is not accessed concurrently and handles one message at a time.

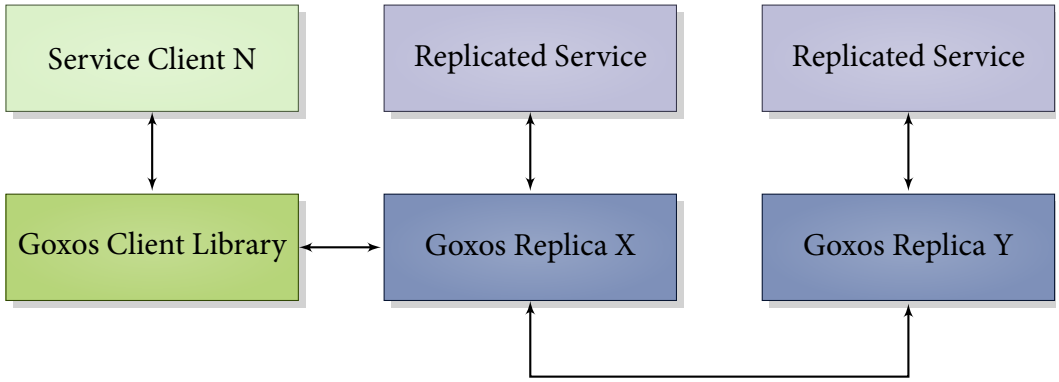

Figure 4.1: Overview of layers for a replicated service using Goxos

Listing 4.1 : Event loop from MultiPaxos Learner

```
1  go func() {
2          for {
3                  select {
4                  case learn := <-l.learnChan:
5                          l.HandleLearn(learn)
6                  case creq := <-l.creqChan:
7                          l.HandleCatchUpReq(creq)
8                  case cresp := <-l.crespChan:
9                          l.HandleCatchUpResp(cresp)
10                 case trustId := <-l.trust:
11                         l.leader = trustId
12                 case <-l.stop:
13                         return
14                 }
15         }
16 }()
```

One of the original goals when designing the Goxos framework was to achieve a highly modular architecture. The main motivation behind this was to enable future extensions to easily and seamlessly to be integrated. Taking this approach has shown itself to be useful throughout the work on this thesis. Two students working on separate Master's theses have been been able to work on the same code base while developing several extension to the framework.

In addition to having a modular design internally, Goxos is also separated from the replicated service. This is in contrast to other Paxos-based implementations such as Zookeeper [17] and Doozer [18], but similar to the approach taken in JPaxos [19]. Such a design introduces no application specific logic in the Goxos framework. Applications using Goxos imports a single package that exposes a simple API for starting and stopping the replicated service. A high-level overview of this design is shown in Figure 4.1.

Since the rest of this thesis will refer to many of Goxos' internal modules, a short description of the relevant ones will be given below. The modules are listed by name

together with their associated package. Figure 4.2 serves as a companion to these descriptions, with a partial set of module interactions shown.

Server (Server)
The Server module is a fundamental part of the Goxos framework. It can be viewed as the central connection hub for the different Goxos modules. The module creates, initializes, starts and stops the other submodules. In addition, it routes client request to the correct Paxos module based on mode of operation. The Server module is the only part of the Goxos framework that interfaces against the replicated service. The Server module invokes the local replicated service to execute decided requests. It also queries the local replicated service for application state during failure handling.

Demuxer (Network)
The Demuxer module handles the incoming network traffic from the other nodes in a Goxos cluster. It decodes incoming messages and routes them to the appropriate submodule. Submodules can subscribe to messages based on message type.

Sender (Network)
The Sender module is responsible for sending messages to the other nodes as requested by other submodules. It maintains a map of connections shared with the Demuxer.

Heartbeat Emitter (Liveness)
The Heartbeat Emitter module broadcast heartbeats at a regular, specified interval. It does not send heartbeats if the running node has recently broadcast another type of message, as every message received is used as an implicit heartbeat.

Failure Detector (Liveness)
The Failure detector module is responsible for providing (possibly inaccurate) information to other submodules about which nodes are believed to have failed. The choice of failure detector is described greater detail in Section 6.1.1.

Leader Detector (Liveness)
The Leader Detector module provides submodules with information about the replica currently assigned the role of coordinator for both the Paxos algorithm and replacement handling (proposer and replacement leader). The leader detector algorithm is discussed in further detail in Section 6.1.2.

Proposer, Acceptor and Learner (MultiPaxos)
These three modules follow the multi-decree Paxos protocol as described in

Section 3.4. In addition, the Learner module is also responsible for initiating and handling a catch-up mechanism. Catch-up is activated if the Learner realize that it has missed decisions for previous slots. The Learner module is also responsible for responding to catch-up requests from other nodes.

Group Manager (Grp)
The Group Manager module encapsulates and centralizes information about the other replicas in a running system. This involves providing the other modules with information such as node identifiers, network addresses and ranks. Some additional details about the Group Manager is provided in Section 6.1.3.

Client Handler (Client)
The Client Handler module handles all communication with clients. It is responsible for accepting and validating client requests, forwarding them to the Server module and sending responses to clients.

Replacement Handler (Lr)
The Replacement Handler extends the Goxos framework to include the Live Replacement failure handing method [3]. The module is responsible for initiating replacements, as well as handling all the messages belonging to the Live Replacement protocol. The module is discussed in detail in Section 6.3.

Reconfiguration Handler (Reconfig)
The Reconfiguration handler allows the Goxos framework to change the set of replicas used to execute the consensus protocol at runtime. The module and its reconfiguration algorithm is discussed in detail in Section 6.2.

Initialization Listener (NodeInit)
The Initialization Listener is a module used by nodes started in *standby* mode. During failure handling, nodes are started in this mode to await joining a running Goxos cluster. The listener runs an initialization protocol that lets a node obtain application state and the relevant configuration settings from a peer. The module is discussed in further detail in Section 6.1.4.

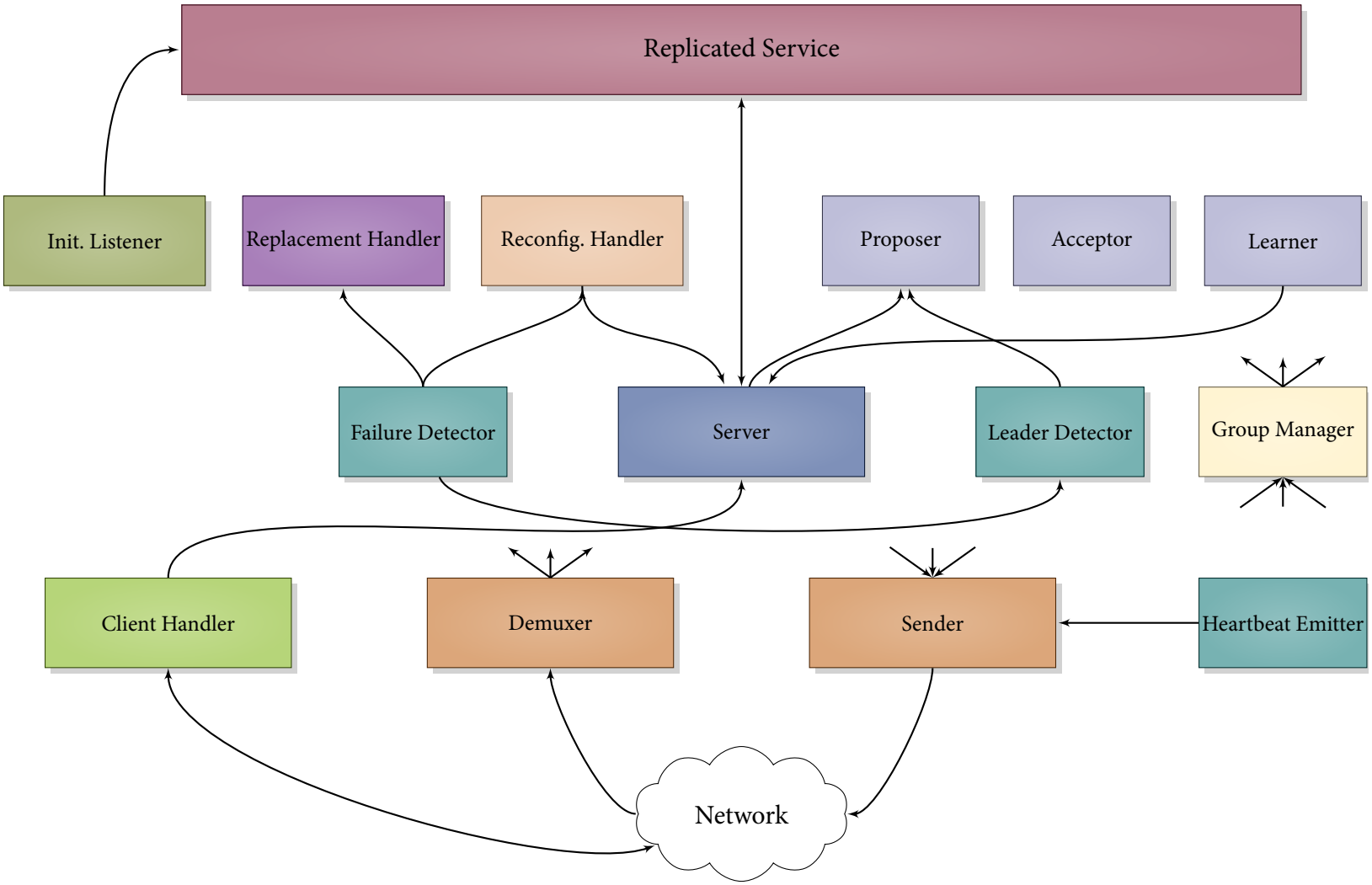

Figure 4.2: Overview of Goxos submodules

4.4 Application Programming Interface

Chapter 6 describes the design and implementation of two failure handling methods for the Goxos framework. This description refers to parts of the Goxos Application Programming Interface (API), available to applications implementing a replicated service. For this reason, a short introduction to the API is given here.

Goxos provides a simple API to applications implementing a replicated service. The API is exposed through the top-level package named Goxos. An application can create two different types of Goxos replicas, either a GoxosReplica or a GoxosStandbyReplica. The constructor function and method signatures available for the two types of replicas are the following:

- GoxosReplica:
  - `func NewGoxosReplica(id uint, appId, configFilePath string, ah app.Handler) *GoxosReplica`
    This function returns a pointer to a new GoxosReplica. The function takes two identifiers as arguments, the replica's PaxosId and a global identifier for the replicated service. The configuration file to be used is specified by a file path. The last argument supplies a handler the Server module use for executing decided requests and requesting application state. The handler is specified through an interface and is described in more detail later in this section.
  - `func (g *GoxosReplica) Init() error`
    This method initializes the replica and returns an error if the procedure fails. The initialization consists of two main steps. The framework first parses and verifies the supplied configuration. If the configuration is valid the replica creates an instance of the Server module, as described in Section 4.3.
  - `func (g *GoxosReplica) Start() error`
    This method starts the replica and returns and error if it fails to do so. The Goxos start procedure involves calling the appropriate start methods of the Server module. The Server module starts the needed submodules based on the supplied configuration.
  - `func (g *GoxosReplica) Stop() error`
    Calling this method attempts to shutdown the replica gracefully, returning an error if any problem is encountered.

- GoxosStandbyReplica:
  - `func NewGoxosStandbyReplica(standbyType string, ah app.Handler, appId, standbyIp string) (gs *GoxosStandby)`
    This function returns a pointer to a new GoxosStandbyReplica. The constructor function differs from the one of a normal replica in a few aspects. It should be noted that no configuration or PaxosId is supplied, since a standby replica obtains this information from the already running cluster during initialization. The `standbyType` argument specifies the standby mode, allowing either Reconfiguration or Live Replacement. Finally, the `standbyIp` provides the IP address of the Initialization Listener.
  - `func (gs *GoxosStandyReplica) Standby() error`
    Calling the `Standby()` method causes the replica to start its Initialization Listener and enter a standby mode. If the replica is initialized, the framework starts the Server module using the settings and state received during the initialization phase.
  - `func (gs *GoxosStandbyReplica) Stop() error`
    This method is the same as for a regular GoxosReplica.

Both constructor functions described above takes an argument, `ah`, of interface type `app.Handler`. This interface definition is shown in Listing 4.2. Each application needs to implement its own appropriate version of this interface. In its current form, the Goxos framework use the supplied implementation of this interface for executing ordered request. In addition, the framework can set and request state from the replicated service using the interface implementation. The interface definition may be expanded in the future for handling other necessary interactions between the Goxos framework and the replicated service.

Listing 4.2 : Application Handler Interface

```
1 type Handler interface {
2         Execute(req []byte) (resp []byte)
3         GetState(slotMarker uint) (sm uint, state []byte)
4         SetState(state []byte) error
5 }
```

5 Failure Handling

This chapter provides an overview of two traditional failure handling methods for Paxos State Machines, namely Recovery and Reconfiguration. In addition, a new method called Live Replacement is introduced. This thesis, as stated in Section 2.1, do not consider or implement recovery, but the method is briefly described here for completeness with regards to failure handling.

5.1 Recovery

In a *fail-recovery* [6, p. 63] model, processes are allowed to crash, but also recover again and participate in the running system. Recovery as a failure handling method has a wide range of applications, ranging from simple program restarts to more complex transactional database recovery schemes [20]. The model normally entails that a recovered process restart on the same machine as its previous incarnation of the process and that the new incarnation of the process have access to the state of its predecessor (obtained from stable storage). Thus this method of recovery fails if the local disk failed, or some other hardware failure preventing the physical machine from restarting the process.

When using Recovery, processes must be able to avoid amnesia, i.e. they should not forget what they did prior to crashing. For a replicated service implemented as a Paxos State Machine this involves writing to stable storage during the protocol, as well as making application state snapshots persistent at regular intervals. This is done to ensure safety across crashes. A recovering replica at startup reads this state

as a part of an initial recovery procedure. A recovered replica may also need to be updated by other replicas of commands decided during its outage.

As opposed to Live Replacement and Reconfiguration, Recovery is able to handle catastrophic failure scenarios, i.e. when more than f replicas fail. This is assumes that fewer than $f + 1$ replicas do not experience disk corruption and that other possible hardware faults may be repaired.

At what steps the Paxos protocol should write to disk to ensure safety is discussed in [21] along with experiments that present the performance impact of disk writes. An approach to efficiently obtaining application state snapshots is described in [15, p. 8]. [19] discusses the different approaches for deciding when to snapshot, in addition to reviewing who should be responsible for initiating it (replica vs. replicated service).

5.2 Reconfiguration

Reconfiguration [5] is a method for changing the set of processes executing a distributed system. Abstractly, for an RSM, this means replacing the current set of replicas (old configuration) with a new set of replicas (new configuration). In addition, to be useful for purposes such as hardware updates and load balancing, Reconfiguration can also be used to exclude and replace failed replicas. The decision to do so can either be initiated by an human administrator or by an appropriate algorithm such as a failure detector.

The classical idea for Reconfiguration (presented in [2]) is to let the replicated service state include the configuration. The system migrates when the old configuration decides on a new sets of replicas and an initial state. The migration is initiated by issuing a special reconfiguration command to the state machine. The command includes the new configuration. That is, the set of replicas to switch to (the new configuration) when the command is learned. Learning this command requires running Phase 2 of multi-decree Paxos, resulting in a Delay of two communication steps. Four communication steps are required if the leader if faulty.

The pipelining parameter described in Section 3.4 introduces more complexity for Reconfiguration when $\alpha > 1$. If $\alpha = 1$, determining the state of the new configuration is simple. When learning the reconfiguration command for slot i, it is known that no other Paxos instance can be running concurrently and all previous requests have been chosen. As a consequence, the Paxos starting state for the new configuration should be obtained from slot $i - 1$. Conversely, if $\alpha > 1$, there may be several consensus instances running concurrently. For handling the remaining $\alpha - 1$ instances after the reconfiguration command, two main approaches as described in [3] will be presented here.

Waiting This approach changes to the new configuration only when all possibly

decided request have been learned. If the reconfiguration command was decided as slot i, the new configuration takes effect from slot number $i + \alpha$. This adds $\alpha - 1$ to Potential Delay. Another variation instructs the Paxos leader to propose null requests[1] for the remaining $\alpha - 1$ instances. Such action can reduce Potential Delay, but increases Disruption since the application requests has to wait for the null request to be decided.

Stopping This approach stops the old configuration directly after deciding on the reconfiguration command. If the command is decided as slot i, decided requests for slots higher than i are discarded. By discarding requests greater than i, Potential Delay is reduced since fewer events requests are required before the new replica can start. However, discarding these request leads to Disruption of at least two communication steps for each request discarded.

Lamport et al. also presents several approaches to Reconfiguration in [5]. A thorough description of an implementation of a waiting Reconfiguration protocol can be found in [22].

5.3 Live Replacement

Live Replacement is a new proposed method for handling failures in a Paxos State Machine. It is, as explained in Chapter 1, one of two failure handling methods implemented and evaluated as a part of this thesis. Live Replacement can, as Reconfiguration, be used to replace a replica that is believed to have failed with a new replica on another machine. As explained in Section 5.2, Reconfiguration relies on a special state machine command for changing the set of replicas used. Live Replacement does not rely on such a command, and it is argued to be independent from state machine progress. As a consequence, a replacement is postulated to have minimal impact on Disruption. The argued independence of Paxos and Live Replacement is formalized using a proposed notion of independent subprotocols.

Live Replacement is introduced and presented by Jehl et al. in "Live Replacement: Fast and Efficient Failure Handling for Paxos State Machines" [3]. This section attempts, using this source, to provide a concise description of the protocol. The reader is referred to the original paper for proof of that Live Replacement do not violate safety of the Paxos State Machine. A discussion of the protocol's liveness properties can also be found here, in addition to the definition and discussion of independent subprotocols. The protocol description given below is specified for single-decree Paxos. Section 6.3 describes the implementation of Live Replacement

[1] Also known as a no-op command.

in the context of multi-decree Paxos. Before presenting a specific replacement example, an overview of protocol actors and messages are given, in addition to other necessary definitions.

The following actors participate in Live Replacement:

Replacement Leader The replica that initiates replacements. The Replacement Leader may also be the leader of the Paxos algorithm, but it is not necessary.

Replacer Replica The new replica replacing an old replica.

Replaced Replica An old replica being replaced.

As stated in Section 2.1, the replicated service considered here consists of n replicas. When applying the Live Replacement method, each replica t is assumed to have an 2-tuple identifier, $id_t = (i_t, e_t)$, where $i_t \in \{0, \ldots, n-1\}$ and $e_t \in \mathbb{N}$. The first element i_t is referred to as the *PaxosId*, while the second element is referred to as the *epoch*. Each replica maintains a vector of epochs, containing the highest known epoch for each PaxosId. The Live Replacement Protocol utilizes a message type containing Paxos state. This is more precisely the state of the Acceptor agent in Paxos. For denoting the Acceptor state $(rnd, vrnd, vval)$ of replica t, Φ_t will be used. It is assumed in this description that a replacement leader knows about spare replicas which can be used to replace faulty ones. Section 6.1.4 later provides details about how this is done in practice.

The theoretical description of the Live Replacement protocol uses two messages for performing a replacement.

- $\langle \textsc{PrepareEpoch}, s, (i_s, e_s) \rangle$: This message is used by a replacement leader to instruct the other replicas to replace the current replica with PaxosId i_s. This replica should be replaced with the new replica s. The message also contains the id of the new replica, (i_s, e_s).
- $\langle \textsc{EpochPromise}, \Phi_t, epoch[]_t, id_t \rangle$: If replica t receives a $\langle \textsc{PrepareEpoch} \rangle$ instructing it to replace a replica with a new one, it sends an $\langle \textsc{EpochPromise} \rangle$ to the new replica containing its acceptor state, epoch vector and id.

To explain the protocol, an example scenario involving a single replacement will be presented. The scenario is also illustrated in Figure 6.4. The system in question consist of three replicas, $n_{2,0}$, $n_{1,0}$ and $n_{0,0}$. Replica $n_{2,0}$ is assumed to be the Paxos Leader, and replica $n_{0,0}$ is assumed to be the Replacement Leader. At some point in time, replica $n_{1,0}$ fails. The Replacement Leader is made aware of this event through either a failure detection mechanism or by some form of human intervention. The

Replacement Leader knows of a replica s that can be used to replace the failed replica.

The Replacement Leader needs to calculate an identifier for s before sending the PrepareEpoch message to the other replicas. For this example, the Replacement Leader calculates the new epoch to be 1, creating the identifier $n_{1,1}$. The Replacement leader finally sends the resulting $\langle \text{PrepareEpoch}, s, (1,1) \rangle$ to all replicas, which in this case is $n_{2,0}$ and itself.

When receiving a PrepareEpoch, the replicas first check that the attached epoch is higher than what it has previously seen for the corresponding PaxosId. In this scenario $n_{2,0}$ approves the PrepareEpoch since $1 > 0$. As a result, it generates and sends an EpochPromise to the new replica s ($n_{1,1}$). The EpochPromise contains its identifier ($n_{2,0}$), epoch vector ($[0,0,0]$) and acceptor state ($\Phi_{n_{2,0}}$). It finally updates the epoch to 1 for PaxosId 1, resulting in the epoch vector $[0,1,0]$. The same procedure is also performed at replica $n_{0,0}$. A replica is said to have *installed a replacer* when it has sent the EpochPromise and stored the new epoch.

The replacer replica $n_{1,1}$ is waiting to receive EpochPromises from a majority of the replicas. When so happens, it extracts a consistent acceptor state from a quorum of messages. For this scenario using single-decree Paxos, this means setting the appropriate rnd, $vrnd$ and $vval$ for the replica's acceptor. The replica adopts the highest rnd seen in the set of EpochPromises, and sets $(vrnd, vval)$ according to the highest seen $vrnd$. The replica afterwards joins the rest of the replicas in running the Paxos Protocol. A replacer replica is said to be *activated* if it has determined an acceptor state from a quorum of EpochPromise messages and started participating in Paxos.

This description now addresses some details and necessary adjustments to Paxos. As stated in the example scenario above, a Replacement Leader needs to calculate a new epoch for the replacer replica. For ensuring the uniqueness of a replacement, the epoch should be generated based on the Replacement Leader's identifier. This is similar to how proposers in the Paxos algorithm employ their own set of unique round numbers.

As stated in Section 2.1, this work assumes a partially synchronous system. It is during asynchronous periods impossible to resolve whether a process has failed or only operating slowly. Live Replacement can for this reason not assume that an old replica has really failed. To handle this fact, the Paxos protocol must be adjusted to ensure that a replaced replica can not be a part of any quorum. Paxos uses two types of messages evaluated in context of quorums, namely Promise and Learn messages. Live Replacement requires a replica to add its epoch vector to every Promise and Learn message. The protocol uses them to ensure that all quorums are valid. The notion of a *Valid Quorum* is defined in [3].

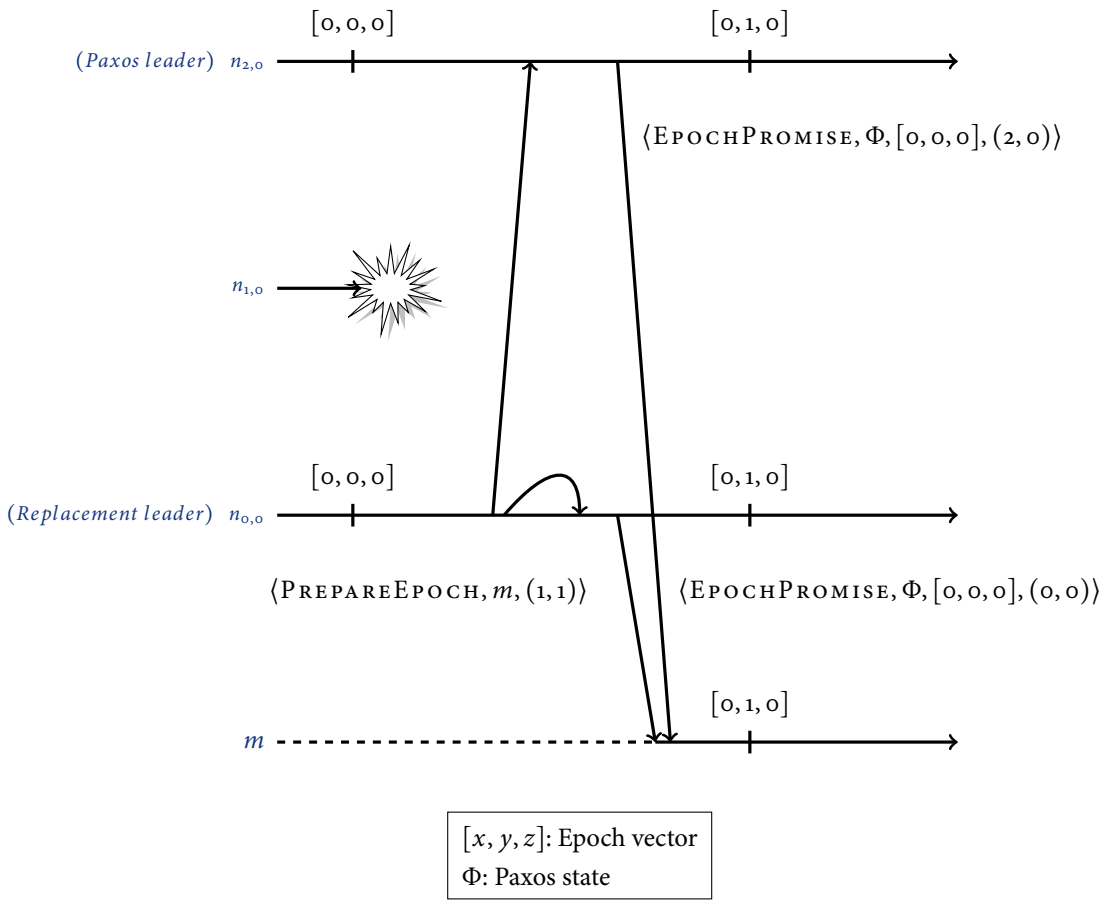

Figure 5.1: The Live Replacement Protocol

Definition. *A quorum of messages* $Q = \{\langle Msg, epoch_t, (i_t, e_t)\rangle, \dots\}$ *is valid if for any pair of messages in* Q *with senders* t *and* t', $i_t \neq i_{t'}$ *and* $epoch_{t'}[i_t] \leq e_t$ *hold.*

In addition to quorums of PROMISE and LEARN messages, Live Replacement also require that quorums of EPOCHPROMISES are valid. This is done to exclude an old replica from the replacement protocol. In addition to valid quorums, old leaders are excluded by having replicas ignore PREPARE messages with an old epoch. This means they ignore a PREPARE from replica t with $e_t < epoch[i_t]$.

As defined in Section 2.1, this work assumes unreliable communication. For this reason, PREPAREEPOCH messages may be be lost and not received by all replicas. A replacer replica solves this by resending the PREPAREEPOCH if it does not receive enough EPOCHPROMISES to determine a vaild quorum. A replacer replica can use a timeout mechanism for deciding if it should resend the PREPAREEPOCH.

Finally, it is mentioned that a system may (due to periods of asynchrony) have several replicas that each considers themselves as the Replacement Leader. If they all try to initiate individual replacements, it may be difficult to obtain vaild quorums of EPOCHPROMISES. A non-activated replacer replica for this reason responds to a PREPAREEPOCH by forwarding the set of all EPOCHPROMISES it has received. The replacer replica associated with the PREPAREEPOCH can then attempt to use these messages for finding a valid quorum. [3] provides a description of such a scenario.

6 Design and Implementation

This chapter describes the design and implementation of two failure handling methods, Live Replacement and Reconfiguration, developed as extensions to the Goxos framework. The chapter is divided into there sections. Section 6.1 gives an introduction and overview of Goxos modules and functioallity that is common for both failure handling implementations. Section 6.2 describes the implementation of the Reconfiguration method, while Section 6.3 presents the design and implementation of the Live Replacement extension.

6.1 Common Modules and Functionality

This section presents two modules and one package common for both implemented failure handling methods. They are presented here since they are often referred to in Section 6.2 and 6.3. The description first focuses on the failure and leader detector modules from the Liveness package. The section ends with a presentation of the NodeInit package.

6.1.1 Failure Detector

Since this thesis concerns failure handling, Goxos' failure detector (FD) is naturally a central component. As stated in Section 4.3, the FD is responsible for providing other submodules with information about which replicas are believed to have failed. This

information may not necessarily be accurate, due to the assumption of a partially synchronous system.

The Goxos FD is loosely based on the "Increasing timeout" algorithm presented in [6, p. 55]. This algorithm implements an eventually perfect failure detector ($\Diamond\mathcal{P}$), capturing the assumptions of a partially synchronous system. Using this algorithm every replica broadcasts heartbeats at a regular interval to signal that they are alive. The FD uses these heartbeats and a repeated timeout procedure to determine the current state of the cluster. The Goxos implementation delegates, as described in Section 4.3 the responsibility of broadcasting heartbeats to a separate hearbeat emitter module.

The FD provides two indication events, ⟨Suspect⟩ and ⟨Restore⟩. Several Goxos modules needs to informed of these events, such as for example the Leader Detector, Live Replacement Handler and Reconfiguration Handler. For disseminating its information to several modules, the FD implements a simple publish-subscribe scheme, allowing modules to register with the FD for receiving indication events. When doing so, a Goxos module is provided with a separate channel on which it can listen for updates.

It is finally noted that the FD indication event ⟨Restore⟩ serves little purpose for the current Goxos implementation. Since the system model assumes only crash failures, such an indication would signal a network partition or a possibly slow replica. As discussed later in this chapter, the Goxos failure handling modules react instantly to ⟨Suspect⟩ indications. This in accordance with the goal of immediate failure handling. The ⟨Restore⟩ event may be used in future adjustments of the framework. In some situations it may be useful to provide replicas with a grace period before considering them faulty, possibly avoiding a costly failure handling operation if a ⟨Restore⟩ is received. A ⟨Restore⟩ can also possibly be used by the failure handling modules to abort a costly replacement procedure before it has reached a point where it is not possible or desirable to do so.

6.1.2 Leader Detector

As mentioned in the description of the Paxos State Machine in Section 3.4, a single replica should be elected as the proposer for the Paxos Protocol. This replica is responsible for proposing incoming requests from clients to acceptors. The replicas can use a leader election primitive to agree on such a common coordinator. Ω is an eventual leader detection abstraction that can be implemented to serve this purpose. The abstraction ensures the uniqueness of the leader only eventually, again capturing the assumptions of a partially synchronous system. Ω has been shown to be the weakest leader detector[1] for solving consensus [23]. Ω can easily be derived

[1] Ω is referred to as a failure detector in literature, but this text will refer to it as a leader detector.

from a $\diamond\mathcal{P}$ failure detector by choosing the leader as the non-suspected process with the highest rank or identifier. The Goxos leader detector (LD) is modeled after the "Monarchical Eventual Leader Detection" algorithm from [6, p. 57], which implements Ω in this fashion using $\diamond\mathcal{P}$.

For a Goxos cluster of n replicas, each replica t has a PaxosId i_t, where $i_t \in \{0, \ldots, n-1\}$. The Goxos LD elects the non-suspected replica with the highest PaxosId to be the Paxos Leader. This is signaled through the LDs only indication event, $\langle \textsc{Trust}, i_t \rangle$, which indicates that the replica with PaxosId i_t is trusted as the Paxos Leader. As for the FD described in Section 6.1.1, the LD also employs exactly the same publish-subscribe pattern for disseminating this information to interested modules. The LD publishes $\langle \textsc{Trust}, i_t \rangle$ messages on each of its subscriber channels.

Section 5.3 introduced the notion of a separate Replacement Leader for the Live Replacement failure handling method. The Replacement Leader could for Goxos be defined to be the Paxos Leader, but it is naturally beneficial to delegate this responsibility to another replica to avoid Disruption. For this reason, the Goxos LD was extended when implementing the Live Replacement method. The LD was adjusted to provide an additional indication event, $\langle \textsc{Trust}', id_t \rangle$, denoting the replica that should be considered as the Replacement Leader. The criteria for choosing the Paxos leader when using Live Replacement was also adjusted to take each replica's epoch intro consideration.

For a Goxos cluster of n replicas employing the Live Replacement method, each replica t has a composite identifier $id_t = (i_t, e_t)$, equal to the one presented in Section 5.3. When employing Live Replacement, the following criteria is used by the Goxos LD for electing the two different leader types:

- **Paxos Leader:** The non-suspected replica with the lowest epoch and the *highest* PaxosId.
- **Replacement Leader:** The non-suspected replica with the lowest epoch and the *lowest* PaxosId.

After some number of replacements, a single replica may end up having the lowest epoch. If this is the case, the replica will take on both leader roles. As stated in Section 5.3 this is not optimal, since the Paxos Leader should be relieved from coordinating any replacement procedure. In the future, a special reconfiguration command may be implemented to enable checkpointing and reseting the replicas identifiers, avoiding such scenarios.

6.1.3 Group Manager

The Group Manager serves as a central repository for Goxos modules, providing information about the replicas of the current cluster. The main source of information is a replica map that encapsulates the identifier, rank and network addresses of the replicas. The replica map is during normal operation only accessed using read requests.

During normal operation, the replica map is only accessed with read operations. The map is only updated during periods of failure handling (a rare event). Normally this would still require that operations on the map be protected with a lock. However, since the majority of operations on the node map are read operations, we wish to avoid such locking. This is achieved by letting modules request to change to node map, thus the Group Manager must ensure synchronized access to the map during this operation. The Group Manager does so by notifying all relevant modules that the map is going to be updated. The Goxos modules that needs consistent membership information must subscribe to hold notifications from the Group Manager. A module is provided with a channel to listen on for such notifications. Upon receiving such a hold request, a module acknowledges the request to the Group Manager, and finally block its event loop. A module continues normal operation when it receives a corresponding release signal from the Group Manager. This synchronization technique relies extensively on use of channels and is not discussed in further detail here.

6.1.4 NodeInit Package

The NodeInit package consists of three main components described in detail in this section:

- Replica Provider Interface.
- Initialization Listener Module.
- Initialization Listener Client Library.

Both the failure handling methods considered in this thesis must be able to obtain new replicas that can be used to replace old ones. This functionality is for Goxos defined as a separate external subsystem, denoted as a Replica Provider. A Replica Provider can be integrated against Goxos by implementing the interface shown in Listing 6.1. The interface contains only one method, `GetReplica`, that returns a value of type `grp.Node`, containing the necessary information about the new replica. The `appId` argument lets the Replica Provider know what kind of replicated service that should be started. The `failureHandlingType` argument specifies what kind

of startup procedure the new replica should use. It is set according to the systems failure handling type.

Listing 6.1 : Replica Provider Interface

```
type ReplicaProvider interface {
        GetReplica(appId, failureHandlingType string) (grp.Node, error)
}
```

To decouple the Replica Provider subsystem from the actual Goxos framework was done for several reasons. Defining this functionality through an interface naturally allow different implementations to be used. To obtain and start a replica is also a procedure that can be tightly connect to the infrastructure the system is running on. Systems may have varying amounts of resources available, and may use specific policies for how to delegate and provide hardware and computing resources. A system may for example take CPU load and other metrics into account when deciding on where to place a new replica. The approach taken attempts to be flexible with respect to what kind of Replica Provider is used. The approach also enables integration against already existing infrastructure systems providing similar services.

The development of Goxos is motivated by the goal of creating a research and testing platform. A very relevant issue in this context would be to integrate the Replica Provider functionality together with the execution of experiments. An experiment platform involves among other thing distributing binaries, injecting failures and collecting log files. All of these tasks can in some form be related to a Replica Provider, since every action needs global knowledge about what replicas are running in the system. The execution of the experiments done for this thesis was not tightly integrated against a Replica Provider. Planned future work for Goxos include developing an experiment platform. Such a platform would likely include a integrated Replica Provider of some form. The Replica Provider used for the experiments considered here were developed as a collaborative side-project. The implementation will not be presented in detail here, but summarized it mainly uses the Secure Shell (SSH) network protocol for deploying and starting replicas at new machines. Future work include adding functionality such as automatic node discovery.

A replica obtained using a Replica Provider is started in a standby mode. This is handled by the replicated service using the Goxos API described in Section 4.4. The replica is in standby mode only running its Initialization Listener module, waiting to be contacted by another replica.

The purpose of the Initialization Listener is to provide a new replica with an identifier, configuration and starting state for the replicated service. The procedure is common for both failure handling implementations discussed later in this chapter. Initialization is performed before the replica begins its startup procedure. This startup procedure differs based on the specific failure handling protocol applied.

The four step initialization protocol is presented below. The description assume that replica t has obtained information about the new replica s through its Replica Provider. Replica t uses the Initialization Listener client library for starting and executing the described protocol.

1. $\langle$InitRequest, $id_t\rangle$: The first message sent from replica t to s is a request with the purpose of determining if the new replica is ready for initialization.
2. $\langle$InitResponse, $ack, state\rangle$: The new replica replies with an boolean ack, indicating if it is ready to be initialized. If ack is false, the $state$ value indicates the current state of the remote Initialization Listener. The $state$ value is currently only being used for logging and debugging purposes, but may be utilized in the future.
3. $\langle$TransferRequest, $id_s, config, \Theta, asid\rangle$: The main transfer is started if t receives a positive acknowledgment from s. The transfer message contains the identifier for s, configuration, application state (Θ) and application state slot identifer ($asid$). This slot identifier denotes what consensus instance the application state is connected to.
4. $\langle$TransferResponse, $success, errorDetail\rangle$: Replica s finally responds to t with a message denoting if the transfer and initialization was successful. The is indicated through the boolean $success$ value. The $errorDetail$ value is used to provide detailed error information if the initialization failed.

The Goxos framework at replica s starts the Server module if the protocol above executes correctly. The startup procedure for the Server module is based on what failure handling method that is applied for the system.

6.2 Reconfiguration

This section presents the design and implementation of the Reconfiguration failure handling method for the Goxos framework. First a description of the implemented Reconfiguration variant is given. The section continues by presenting the set of Goxos modules relevant for Reconfiguration and concludes by describing parts of the implementation using a single failure scenario.

6.2.1 Variant: Waiting Reconfiguration

The reconfiguration variant implemented is based on the *waiting* approach described in Section 5.2. The Goxos implementation of multi-decree Paxos uses pipelining (see Section 3.4), and lets the α parameter be specified as a configuration option.

As described in Section 5.2, the reconfiguration implementation needs to handle the remaining $\alpha - 1$ instances after the decided reconfiguration command. The Goxos framework proposes client requests for these remaining instances. An overview of the reconfiguration protocol used is shown in Figure 6.2. The figure contains two implementation specific messages, ⟨FirstSlot⟩ and ⟨Join⟩, that are explained in more detail throughout this section.

The motivation for implementing Reconfiguration for Goxos were to implement a classical variant of method as described in the literature. The goal was to create a baseline implementation that could be used for testing and comparison against Live Replacement. There exist more comprehensive implementations of Reconfiguration (for example [22]), that to a larger extent addresses corner cases and other subtle details. Our implementation of Reconfiguration follows the waiting variant described in [2].

6.2.2 Goxos Modules

The main Goxos component initiating and handling a reconfiguration is the Reconfiguration Handler module introduced in Section 4.3. This section provides a more detailed description of the Goxos modules relevant to Reconfiguration. The overall description focus on the responsibility and behavior of each module. The presentation is divided into two parts. The first part presents the Goxos modules from the point of view of the already running replicas, including the Paxos leader that initiates a reconfiguration. The second part presents the modules in the context of a new replica that joins a running cluster as part of a new configuration. An overview of the Goxos modules relevant for Reconfiguration is show in Figure 6.1. The figure serves as a companion to the presentation given below.

For an already running replica, the module overview is as follows:

Failure Detector
The FD is responsible for informing the Reconfiguration Handler about any suspected replicas.

Leader Detector
Described in the context of Reconfiguration, the LD is responsible for informing the Reconfiguration Handler about the current Paxos leader. The Reconfiguration Handler described below will only react to failure indications if the running replica is considered to be the Paxos leader.

Proposer and Learner
Since a reconfiguration command needs to be proposed and learned, both the Proposer and Learner modules from the MultiPaxos package are involved in performing a reconfiguration procedure.

Reconfiguration Handler
The Reconfiguration Handler is the main component for the reconfiguration implementation. The module initiates a reconfiguration procedure based on input from the FD. It is also responsible for initializing a new replica. The module creates the reconfiguration command that specifies the new configuration and forwards it to the Proposer through the Server module.

Server
The Server module is mainly responsible for ensuring correct behavior of a replica from a reconfiguration command is learned and until the new configuration takes effect. This involves keeping track of when the replica should switch to the new configuration. The Server module can do this since it executes decided requests received from the Learner module. In addition, the module routes client requests to the Proposer, enabling it to control that requests are proposed correctly for the $\alpha - 1$ instances after the reconfiguration command. The module also obtain application state from the replicated service when the Reconfiguration Handler requests it.

Sender and Demuxer
These two modules from the NETWORK package is naturally involved for sending the receiving messages related to the reconfiguration procedure since a reconfiguration command needs to be both proposed and learned.

Group Manager
The Group Manager is used by the Server module to update the node map when changing to a new configuration.

In addition to the modules above, the Reconfiguration Handler uses the provided client library in the NODEINIT package (described in Section 6.1.4) for contacting a new replica's Initialization Listener.

For a new replica started as part of a reconfiguration procedure, the module overview is as follows:

Initialization Listener
This module is used by the new replica to obtain its initialization data, as described in Section 6.1.4. When using Reconfiguration, the new replica receives its configuration and application state from the Paxos leader.

Demuxer
The Demuxer is in the initial replica pre-start phase responsible for forwarding received ⟨FIRSTSLOT⟩ and ⟨JOIN⟩ messages to the Reconfiguration Handler.

Reconfiguration Handler
For a new replica waiting to join a new configuration, the Reconfiguration Handler is responsible of forwarding any received ⟨FirstSlot⟩ or ⟨Join⟩ message to the Server module.

Server
The Server module is started after the Initialization Listener successfully has obtained and set the state of the replicated service. The Server is responsible for starting the other relevant Goxos modules. Starting the modules is triggered by receiving ⟨FirstSlot⟩ and ⟨Join⟩ messages from the Reconfiguration Handler.

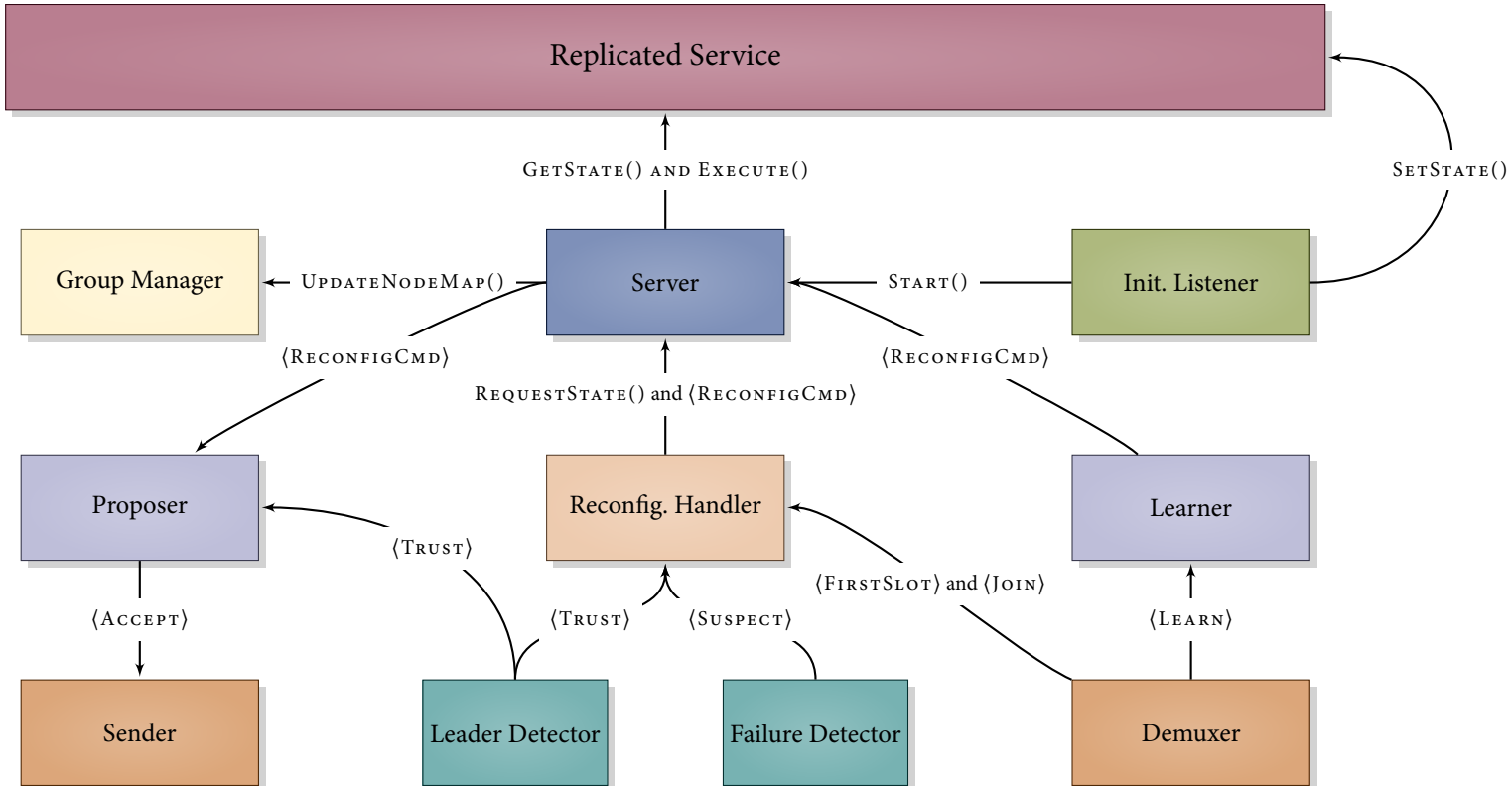

Figure 6.1: Goxos modules relevant for Reconfiguration

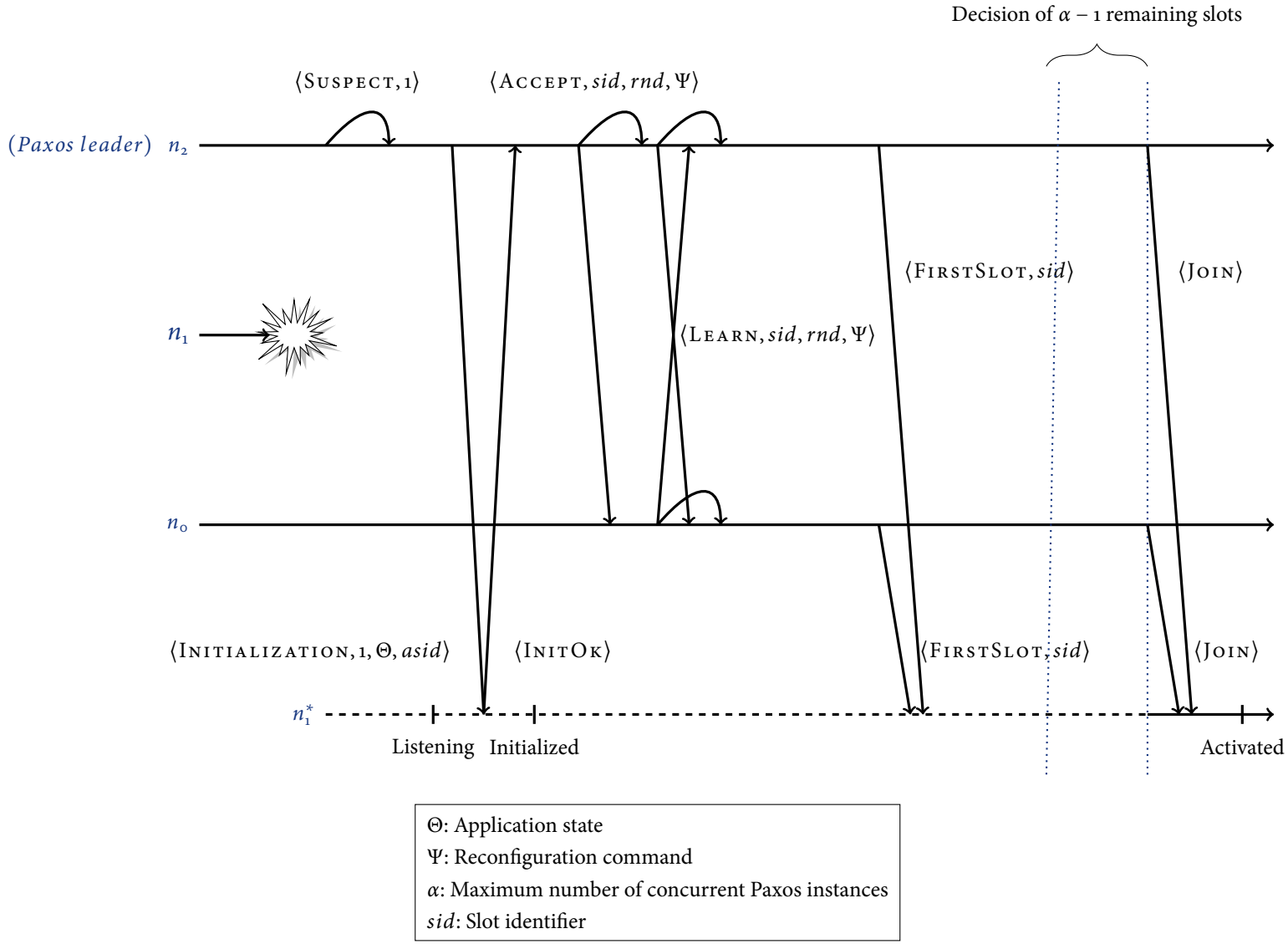

Figure 6.2: Reconfiguration due to failure of a single node

6.2.3 Failure Scenario

Here follows a presentation of how a reconfiguration is performed using the current implementation. The single failure scenario shown in Figure 6.2 serves as the basis for this description. The system considered here consist of three replicas, n_0, n_1 and n_2. The new replica joining as part of the new configuration will before it has been assigned a PaxosId be referred to as replica m. Replica n_2 is assumed to be acting as the Paxos leader, and Phase 1 of multi-decree Paxos is assumed to be completed. The system is defined to be synchronous for the whole duration of this scenario.

The description below refers to a set of implementation specific variables and definitions which will be explained first. A new replica joining a result of a reconfiguration transition through four defined states. These state definitions will be used in the scenario below, and also when describing experimental results in Chapter 7. The four states are:

- **Listening**: The new replica has started and is listening for initialization data.
- **Initialized**: The new replica has obtained its identifier, configuration and replicated service state.
- **Prepared**: The new replica has received the necessary information to be able to initalize and start its Paxos modules.
- **Activated**: The new replica has joined the Goxos cluster as part of the new configuration and is participating in the Paxos protocol.

The set of implementation specific variables referred to below are all of type `paxos.SlotId`. They denote a specific slot of the mulit-decree Paxos protocol.

- `adu`: Variable maintained by the Server module, denoting the highest decided slot it has executed (all decided up to).
- `nextSlot`: Variable belonging to the Proposer, used to keep track of which slot it should tie its next proposal to.
- `nextExpectedDecided`: Variable maintained by the Learner, denoting the next expected slot to be decided. The Learner starts a catch-up mechanism if it learns a decision for a slot higher than this value.

The failure scenario considered here is as follows:

1. At some point in time, replica n_1 fails by crashing.
2. The Reconfiguration Handler at n_2 is informed about this event by the FD through a $\langle \textsc{Suspect}, n_1 \rangle$ message.

3. The Reconfiguration Handler reacts to this indication event by trying to initialize a new replica. First the module request a new replica from the system's Replica Provider. The Reconfiguration Handler is provided with replica m as a response. Replica m is *listening*, and started in reconfiguration standby mode, as explained in Section 6.1.4.

4. The Reconfiguration Handler at n_2 prepares the initialization of replica m by collecting the necessary data. This includes creating a new replica map that includes the new replica m. Additionally, the current system configuration is generated. Finally, the Replacement Handler request application state and a corresponding slot marker from the Server module. The Server module forwards this request to the replicated service. The service replies to the Reconfiguration Handler directly, using a separate channel.

5. The Reconfiguration Handler starts, when it has obtained all the necessary information as explained above, the four-step initialization protocol described in Section 6.1.4. This protocol is show in Figure 6.2 as a single exchange of $\langle \textsc{Initialization}, \Theta, asid \rangle$ and $\langle \textsc{InitOk} \rangle$, where Θ denotes application state and $asid$ the identifier for the consensus slot connected with the application state.

6. If the initialization procedure for the new replica m was successful, m starts its network modules to be able to receive the messages exchanged as part of the reconfiguration protocol. The new replica is now also aware of its PaxosId, n_1^*, and the system configuration, making it *initialized*. Before taking further action, the replica waits to receive a $\langle \textsc{FirstSlot} \rangle$ message, indicating the first Paxos slot in the new configuration.

7. At the Paxos leader, n_2, the Reconfiguration Handler creates the reconfiguration command, here denoted as Ψ. The command includes the new replica map created in step 4. After creation the command is sent to the Server module.

8. The reconfiguration command is at replica n_2 forwarded from the Server module to the Proposer module. As a result, the Proposer broadcasts an $\langle \textsc{Accept}, sid, rnd, \Psi \rangle$ to all replicas. The message contains a slot identifier, sid, which is used in multi-decree Paxos to denote the consensus instance.

9. The next phase of the reconfiguration procedure begins when the reconfiguration command is decided by a quorum of the replicas. The Server module at replica n_2 and n_0 enters a special transition period upon receiving a decided reconfiguration command, waiting for the new configuration to become activated. The replicas (with the exception of the Paxos leader) only

allow execution of decided requests during this period. After executing the reconfiguration command replica n_2 and n_0 is able to calculate the first slot of the new configuration. The first slot of the new configuration will be (*after* incrementing the local `adu`) `adu + alpha`. Both replicas now initiate a TCP connection to the new replica and send first slot identifier using a ⟨FIRSTSLOT⟩ message. Afterwards, both n_2 and n_0 waits for the decision of $\alpha - 1$ client requests that may be in the pipeline. As the Paxos leader, replica n_2 also needs to propose enough client requests to fill the gap of the $\alpha - 1$ instances needed before the new configuration can take effect. The Server therefore listens for incoming client requests from the Client Handler module. The Server module allows, in accordance with the *waiting* reconfiguration variant described in Section 5.2, $\alpha - 1$ requests to be forwarded to the Proposer, before waiting for enough decided slots, making it able to switch to the new configuration.

10. Replica n_1^* is now able to initialize its Paxos modules when it receives a ⟨FIRSTSLOT⟩ message from one of the other replicas. The replica's `adu` is set to the slot marker that was bundled with the application state received during the initialization phase. The Proposers `nextSlot` variable is set to the value contained in the ⟨FIRSTSLOT⟩ message. The Learner is initialized with its `nextExpectedDecided` variable set to `adu + 1`. This is important for ensuring that the catch-up mechanism is triggered after joining the other replicas in executing the Paxos protocol. After initialization the Paxos modules are also started, resulting in that n_1 now is *prepared*.

11. After executing slot with identifier `firstslot - 1`, the Server Module installs the new configuration. The Server instruct the other modules to pause their operation while replacing the Group Manager's replica map, using the functionality described in Section 6.1.3. After the new replica map is installed, each replica attempt to connect to n_1^*. Each replica sends n_1^* a ⟨JOIN⟩ message when they establish a connection. The replicas afterwards resume normal operation. Both ⟨FIRSTSLOT⟩ and ⟨JOIN⟩ messages are sent by every replica to n_1^*. If a single replica is delegated this responsibility, it may fail before sending any of the two messages. To address this, every replica sends these messages. A new replica only uses the first ones received.

12. The first ⟨JOIN⟩ received by replica n_1 is used as an indication for that the new configuration has started. The replica can now start its remaining modules, including the Heartbeat Emitter, Client Handler, FD and LD. Replica n_1^* is now *activated*.

The ⟨JOIN⟩ message used in the implementation described above may be claimed to be unnecessary. One could argue that a new replica could intercept either a

⟨HEARTBEAT⟩ or any Paxos message sent to it, and use that as signal for when to join the running cluster. This is true from a theoretical point of view, but doing this intercept with the current version of Goxos would complicated implementation extensively. The implementation may be changed to use this approach in the future, but the changes needed for the network modules were not prioritized work for this thesis.

6.3 Live Replacement

This section presents the design and implementation of the Live Replacement failure handling method. The Goxos modules relevant for Live Replacement is presented first in context of a single failure scenario. Then a detailed description of some technical details related to the implementation is provided.

6.3.1 Goxos Modules

The main component initiating and executing a replacement is the Replacement Handler introduced in Section 4.3. This section provides an module overview, similar to presentation given for the Reconfiguration implementation. The description is divided into two parts, one from the perspective of the non-failed replicas and one from the view of a replacer replica. Figure 6.3 provides an overview of the Goxos modules relevant for Live Replacement.

For a normal replica and the replica acting as the Replacement Leader, the following modules are involved in Live Replacement:

Failure Detector
The FD is responsible for informing the Replacement Handler about any suspected replicas.

Leader Detector
The LD is responsible for informing the Replacement Handler about the current Replacement Leader. The Replacement Handler will only react to failure indications if the running replica is considered to be the Replacement Leader.

Replacement Handler
The Replacement Handler is responsible for initiating and executing replacements based on indications from the FD. Executing a replacement includes obtaining and initializing a new replacer replica, followed by broadcasting a ⟨PREPAREEPOCH⟩ message to the other replicas. Upon receiving a ⟨PREPAREEPOCH⟩ message the Replacement handler installs the replacer and responds with an ⟨EPOCHPROMISE⟩.

Acceptor
The Acceptor module is accessed by the Replacement Handler when a replica generates a ⟨EpochPromise⟩ in response to a ⟨PrepareEpoch⟩ message. The Acceptor is queried for its Paxos State by the Replacement handler. This state is attached in the ⟨EpochPromise⟩ sent to the new replacer replica.

Server
The Server module is responsible for forwarding state transfer requests from the Replacement Handler to the Replicated Service.

Group Manager
The Group Manager is accessed by the Replacement Handler when installing a new replica. The Group Manager also encapsulates each replicas epoch vector.

In addition to the modules above, the Replacement Handler uses the provided client library in the NodeInit package to contact Initialization Listener of a replacer replica.

For a replacer replica, the module overview is as follows:

Initialization Listener
The Initialization Listener serves the same purpose as for a replica starting as part of a reconfiguration, see Section 6.2.2.

Replcament Handler
At a replacer replica the Replacement Handler receives ⟨EpochPromise⟩ messages, and attempts to extract a consistent Paxos State from the received set of ⟨EpochPromise⟩ messages. If the Replacement Handler obtains a consistent Paxos state, it instructs the Acceptor module to adopt this state and sends an activation signal to the Server module.

Server
The Server is responsible for initializing and starting the appropriate modules both before and after activation of a replacer replica.

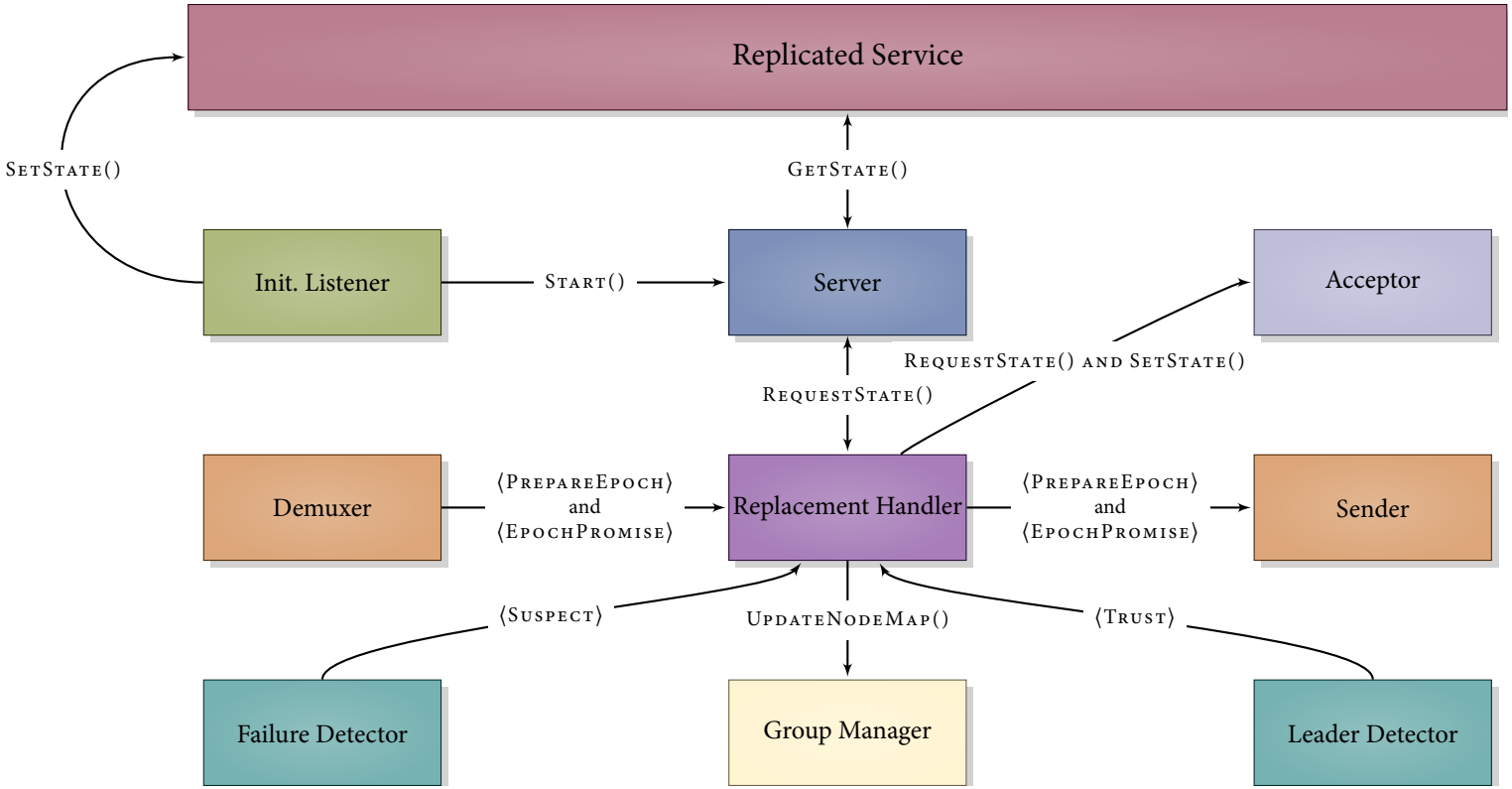

Figure 6.3: Goxos modules relevant for Live Replacement

6.3.2 Failure Scenario

As for the Reconfiguration, the Live Replacement implementation is also described through an example failure scenario. The single failure scenario in Figure 6.4 serves as the basis for this description. The cluster consist of three replicas, $n_{2,0}$, $n_{1,0}$ and $n_{0,0}$. The replacer replica joining the cluster as part of a replacement is referred to a m before it is assigned its identifier. Replica $n_{2,0}$ is assumed to by acting as the Paxos Leader and $n_{0,0}$ as the Replacement Leader. The system is assumed to be synchronous for the whole duration of this scenario.

A new replica joining as a result of a replacement procedure will transition through three defined states. These state definitions is used in the scenario below, and also when describing experimental results in Chapter 7. The three states are:

- **Listening**: The new replica is started and listening for initialization data.
- **Initialized**: The new replica has obtained its identifier, configuration and replicated service state.
- **Activated**: This definition follows from the theoretical description (see [3] and Section 5.3). An activated replacer replica has determined a Paxos State from a valid quorum of $\langle$EpochPromise$\rangle$ messages, and has started participating in the Paxos protocol.

The failure scenario considered here is as follows:

1. At some point in time, replica $n_{1,0}$ fails by crashing.
2. The Replacement Handler at $n_{0,0}$ is informed about this event by the FD through a $\langle$Suspect, $n_{1,0}\rangle$ message.
3. The Reconfiguration Handler reacts to this indication event by initializing a new replica. Firstly, the module request a new replica from the system's replica provider. The Replacement Handler is provided with replica m as a response. Replica m is *listening*, and started in replacement standby mode, as explained in Section 6.1.4.
4. The Replacement Handler at $n_{0,0}$ prepares the initialization of m by generating its identifier. As explained in Section 5.3, this involves reusing the PaxosId for the failed replica and computing a new epoch. For this scenario the epoch is 1, resulting in the identifier $n_{1,1}$. How this epoch is computed is explained in Section 6.3.4 below. The Replacement leader finally executes the initialization of replica m using the same procedure as described in step 4 and 5 in Section 6.2.3.

5. If the initialization procedure for replacer replica m is successful, m starts its network modules, Heartbeat Emitter and Replacement Handler module. In its current state the new replica is aware of its identifier, $n_{1,1}$, and the system configuration, making it *initialized*. Before taking further action, the replica waits to receive a valid quorum of ⟨EpochPromise⟩s.

6. Following a successful initialization of $n_{1,1}$, the Replacement Handler at $n_{0,0}$ broadcasts a $\langle \text{PrepareEpoch}, n_{1,1}, m, asid \rangle$. The necessary network addresses are denoted by m and $asid$ denotes the slot identifier associated with the application state initialized at $n_{1,1}$

7. Upon receiving the ⟨PrepareEpoch⟩, $n_{2,0}$ and $n_{0,0}$ checks if the epoch is larger than it has previously seen for PaxosId 1. Both replicas accept to handle the ⟨PrepareEpoch⟩ since $1 > 0$. As a result, both replicas initiate a TCP connection to $n_{1,1}$. Replicas $n_{2,0}$ and $n_{0,0}$ now generates an ⟨EpochPromise⟩ message according to the Live Replacement protocol, containing their identifier, epoch vector and acceptor state. Generating the ⟨EpochPromise⟩ involves several implementation specific details, and is therefore described in detail in Section 6.3.3. The ⟨EpochPromise⟩ is here assumed to be generated, causing each replica to send the ⟨EpochPromise⟩ message to $n_{1,1}$. Finally, both $n_{2,0}$ and $n_{0,0}$ request the Group Manager to replace replica $n_{1,0}$ with $n_{1,1}$. This also causes the Group Manager to update a replica's epoch vector accordingly. Both $n_{2,0}$ and $n_{0,0}$ is now said to have *installed* the replacer.

8. The replacer replica is waiting to obtain a valid quorum of ⟨EpochPromise⟩ messages. Upon receiving such a message, the replica stores it and check if it potentially can verify a valid quorum, meaning it has stored at least a quorum of messages. If this is the case, the Replacement Handler tries to verify a valid quorum. The procedure for verifying a valid quorum is described in Section 6.3.5. If the replacer is able to verify a valid quorum, the replica proceeds to extract a consistent acceptor state. The replica starts by generating an empty acceptor slot map. For every slot seen in the set of valid ⟨EpochPromise⟩ messages, the replica adopt the $(vrnd, vval)$ with the highest rnd, and store them in the slot map. The Replacement Handler then instructs the Acceptor to adopt this slot map. The Replacement Handler finally signals the Server module to activate the replica by starting the remaining modules. These are the Paxos modules, LD, FD and Client Handler. The replacer replica is now *activated*.

In the following sections the complications involved in implementing steps 7, 4 and valid quorum verification will be discussed in detail.

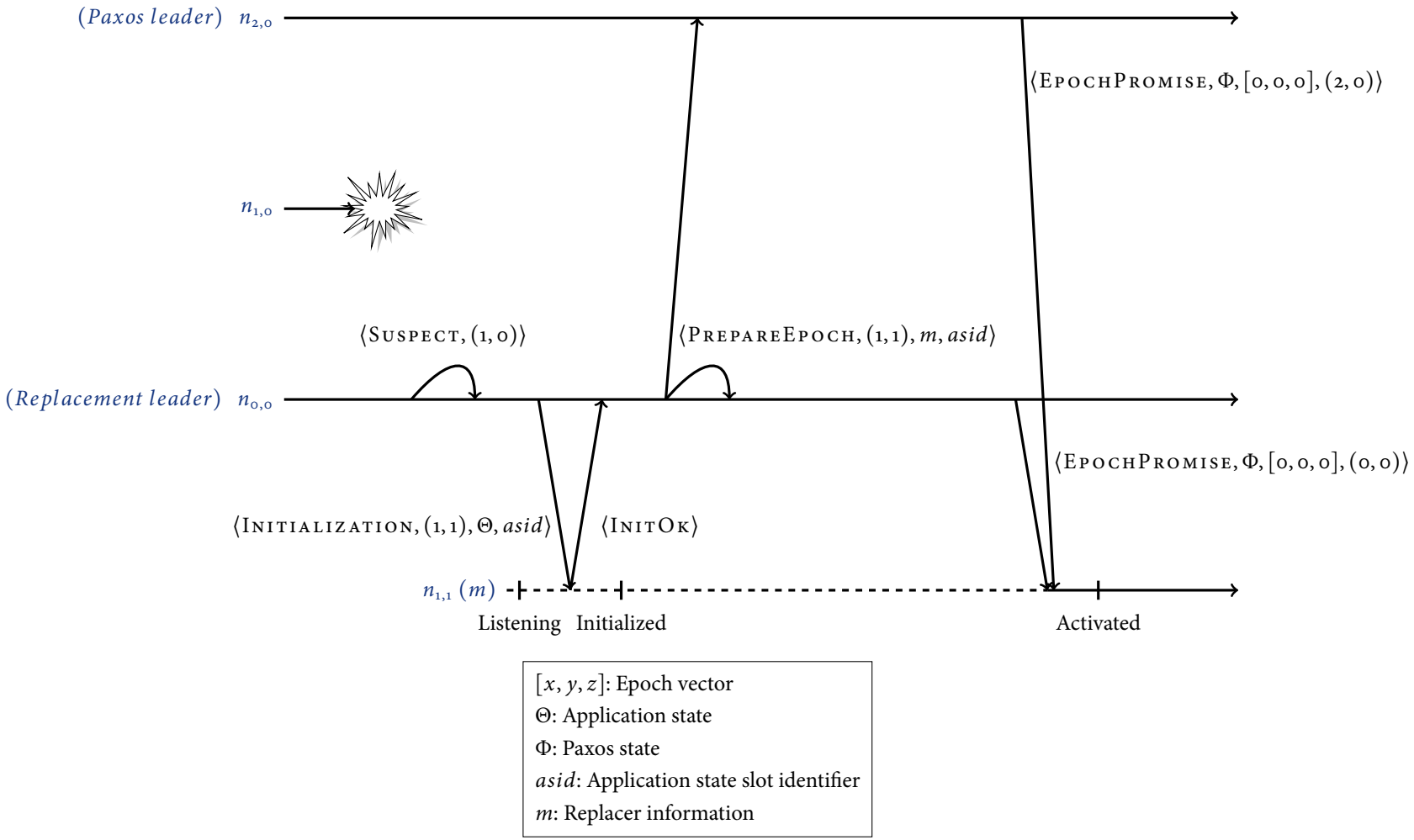

Figure 6.4: Replacement of a single failed node

6.3.3 PrepareEpoch Handling

The section provides a detailed description of step 7 in the failure scenario above. This step includes generating and sending an ⟨EpochPromise⟩, causing a replica to install a replacer. This more detailed description of this step is provided due to the extra synchronization needed between the Replacement Handler and Acceptor module. Additionally, describing the related implementation serves as a concise example of how using channels can achieve this synchronization instead of using traditional locking of shared state.

In step 7, a replica handles the ⟨PrepareEpoch⟩ if the epoch is larger than what it has previously seen for the given PaxosId. The replica generates and sends an ⟨EpochPromise⟩ to the replacer replica if this is the case. It finally install the replacer by updating its replica map and epoch vector. The ⟨EpochPromise⟩ is sent before the epoch vector is updated to enable valid quorums that may include a replacer replica. Generating and sending the ⟨EpochPromise⟩ should be executed as an atomic operation, i.e. it should either successfully complete or have no effect. The operation should also be ensured isolated from concurrent running Goxos modules (goroutines). This requires the Acceptor module to hold its operation until it is notified that the replacer is installed. The reason for this is that the ⟨EpochPromise⟩ should only contain Acceptor state corresponding to the replica's current epoch vector. This is ensured by instructing the Acceptor temporarily stop participating in the Paxos protocol until the replacemnt is executed. The Group Manager should also hold the other relevant modules during this procedure, since installing a replacer involves updating both the node map and epoch vector. How this atomic step is implemented is explained below.

First the Replacement Handler requests the Group Manager to pause the operation of relevant Goxos modules using the procedure described in Section 6.1.3. Continuing, the Replacement Handler need to query the Acceptor module for the Paxos state, using the `GetState` method shown in Listing 6.2. The Acceptor in the Goxos framework is defined through the interface shown in Listing 6.2, and the relevant `GetState` method take two arguments. The `afterSlot` argument of type `SlotId` indicate that the caller only wants state for slots after this slot identifier. The `release` argument is a receive only channel with a boolean data type. The Acceptor must block on a read operation on this channel after returning the requested state to the Replacement Handler. The channel is used by the Replacement Handler to signal that the Acceptor can continue operation.

Listing 6.2 : Acceptor interface

```
type Acceptor interface {
        PaxosActor
        HandlePrepare(PrepareMsg) error
        HandleAccept(AcceptMsg) error
        GetState(afterSlot SlotId, release <-chan bool) *AcceptorSlotMap
        SetState(slotmap *AcceptorSlotMap)
}
```

Listing 6.3 shows how the Replacement Handler calls the `GetState` method of its associated acceptor module. As described above, the ⟨PrepareEpoch⟩ contains an *asid* value, denoting the corresponding Paxos slot identifier for the replacer's initialized application state. The Replacement Handler uses this value (`pe.SlotMarker`) as the `afterSlot` argument. The module also creates a unbuffered channel, `accRelease`, that it uses to signal that the acceptor can continue. As shown in Listing 6.3, the Replacement Handler calls the built-in function `close` on the `release` channel directly after creating it. This is done using a `defer` statement. A defer statement "invokes a function whose execution is deferred to the moment the surrounding function returns" [14]. This means the `accRelease` channel will be closed when the `handlePrepareEpoch` method returns, regardless of if installing the replacer succeed or not. The Acceptor module blocking on a receive operation on this channel will be released when the `handlePrepareEpoch` method returns. This is because closing a channel will cause a receive operation to return the zero value for the channel's type without blocking. It should be noted that no actual boolean value is sent over the channel. The channel's boolean type is merely chosen for it is a small sized data type.

After receiving the `acceptorSlotMap` the Replacement Handler is able to generate its ⟨EpochPromise⟩. After sending the ⟨EpochPromise⟩ the Replacement Handler tells the Group Manger to update its replica map and epoch vector. The Acceptor is finally released when the `handlePrepareEpoch` method returns, due to using the `defer` statement for closing the signaling channel as explained above. The Goxos modules earlier instructed by the Group Manager to hold operation are also released using a similar mechanism.

Listing 6.3 : Requesting state from the Acceptor module

```
func (rh *ReplacementHandler) handlePrepareEpoch(pe PrepareEpoch) {
        // ...

        accRelease := make(chan bool)
        defer close(acceptorRelease)
        acceptorSlotMap := rh.acceptor.GetState(pe.SlotMarker, accRelease)

        // ...
}
```

6.3.4 Epoch Generation

Step 4 of the failure scenario described in Section 6.3.2 involves calculating an epoch for the replacer replica. It is important that a generated epoch is unique, since during periods of asynchrony there may be multiple replacement leaders issuing replacements for the same PaxosId. Although unlikely, it may happen that an old and new replica initiates replacements. For these reasons, a replacement leader t should generate a replacer epoch based on both i_t and e_t. The theoretical description of Live Replacement [3] mentions the uniqueness requirement, but give no specific instructions on how to compute epochs. A description of how this is done for the Goxos implementation is therefore given below.

The Goxos Live Replacement implementation uses a pairing function to generate unique epochs. A paring function [24] is an operation to uniquely encode two natural numbers into a single natural number. The implementation employs the Cantor paring function, which is a bijective function, $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, defined as [25]:

$$\pi(k_1, k_2) := \frac{1}{2}(k_1 + k_2)(k_1 + k_2 + 1) + k_2 \tag{6.1}$$

The implementation uses this function twice to generate an epoch, mapping from three natural numbers to a single natural number. For a replica t, the first two numbers are i_t and e_t, from the composite identifier id_t used when employing Live Replacement. The third number is a local counter, n_t, incremented as necessary. This procedure is essentially applying a generalized definition of Equation 6.1 above, called the Cantor tuple function, $\pi^{(n)} : \mathbb{N}^n \to \mathbb{N}$. It is defined as [25]:

$$\pi^{(n)}(k1, \ldots, k_{n-1}, k_n) := \pi(\pi^{(n-1)}(k_1, \ldots, k_{n-1}), k_n) \tag{6.2}$$

Thus, the procedure to generate an epoch at the Replacement Leader t, for replacer replica s replacing replica r, can be defined as:

$$e_s = f(id_t, e_r) = min\{e \in \pi^3(i_t, e_t, n_t) | e > e_r\} \tag{6.3}$$

How this function is implemented is shown in Listing 6.4. The Replacement Handler at initialization obtain an epoch generator function by calling the `getEpochGenerator` function. The function takes the composite identifier of the replica as an argument. The function returns a new epoch generator function which takes an epoch as argument and returns an epoch. This returned function is a closure[2], capturing two variables. These variables are the local counter, `n`, and a epoch

[2] A closure is a function or reference to a function together with a referencing environment. The closure captures, if it refers to them, any constants or variables present in the same scope where it is created [26].

generation base constant, `baseForNode`. The `baseForNode` is only calculated once using the Cantor paring function. The returned generator function takes a replaced replica's epoch, `oldEpoch` as an argument. The generator function then applies the Cantor paring function with `n` and `baseForNode` as arguments. This is done, incrementing `n` as necessary, until the result is higher than the `oldEpoch` argument.

Listing 6.4 : Epoch Generation

```
1  package lr
2
3  import "goxos/grp"
4
5  func getEpochGenerator(id grp.Id) func(grp.Epoch) grp.Epoch {
6          var n uint64 = 0
7          baseForNode := cantorPairing(uint64(id.PaxosId), uint64(id.Epoch))
8          return func(oldEpoch grp.Epoch) grp.Epoch {
9                  var replacerEpoch uint64
10                 for replacerEpoch <= uint64(oldEpoch) {
11                         replacerEpoch = cantorPairing(baseForNode, n)
12                         n++
13                 }
14
15                 return grp.Epoch(replacerEpoch)
16         }
17 }
18
19 func cantorPairing(k1, k2 uint64) uint64 {
20         return (k1+k2)*(k1+k2+1)/2 + k2
21 }
```

6.3.5 Valid Quroum Verification

In step 8 of the failure scenario described in Section 6.3.2, a replacer replica attempts to verify a valid quorum from its set of ⟨EpochPromise⟩ messages. How this verification is implemented is described here. A replacer replica uses the procedure shown in Algorithm 1 and follows a brute-force approach. Summarized, if the quorum size for the system is q, the algorithm iterates through every possible q-sized combination from a set of n messages. A combination of messages that form a valid quorum is returned immediately. The algorithm return an empty set of messages if no valid quorum is found. The procedure has a worst-case running time of $\mathcal{O}(\binom{n}{q}q^2)$.

There clearly exist optimization possibilities for the procedure discussed here. For instance, some duplicate computation is performed when checking pairs of messages for conflict. A better approach could be to implement a variant using depth-first search with backtracking to reduce comparisons. However, spending time on such optimization has not been a priority during this work, since the algorithm is only employed during periods of asynchrony, when non-faulty replicas could get replaced without being aware of it. Additionally, the size of the input to the

algorithm, n and q, can for a typical system deployment be assumed to be relatively small.

6.3.6 Paxos Adjustments

As mentioned in Section 5.3, also ⟨PROMISE⟩ and ⟨LEARN⟩ messages need to be valid when using Live Replacement. The cost of running Algorithm 1 is high, and thus it is undesirable to use it for verifying every received ⟨LEARN⟩ message. Therefore the implementation uses two levels of valid quorum verification, *simple* and *extended*. Simple valid quorum verification requires that a replica checks every received message's epoch vector against its own, and only when seeing a differing epoch vector switches to extended valid quorum verification. The extend verification is the one shown in Algorithm 1. The Learner module performs the simple verification for every ⟨LEARN⟩ message received, and only if it sees differing epoch vectors will the Learner module invoke the extended verification algorithm. The extended verification is only enabled for the specific *slot* the ⟨LEARN⟩ message is tied to. The Learner marks slots that need extended verification so that only those are verified using the extended procedure. The Proposer module need to verify a valid quorum during phase 1 of the multi-decree Paxos protocol. Switching to extended verification for this module is therefore tied to phase 1 itself, and not a specific slot. Finally in Section 8.2 it is discussed how a future implementation may avoid the simple valid quorum verification altogether.

Algorithm 1 Verifying a valid quroum

Input:
M { Set of n messages of equal type, either ⟨PROMISE⟩, ⟨LEARN⟩ or ⟨EPOCHPROMISE⟩ }
q { Quorum size }

Output:
vq { Boolean indicating if a valid quorum is found }
C { A q-sized set of messages from M forming a valid quorum }

Definition:
Messages msg_j with $(i_j, e_j, epoch_j[])$ and msg_k with $(i_k, e_k, epoch_k[])$ are in conflict if
$(i_j = i_k) \vee (epoch_j[i_k] > e_k) \vee (epoch_k[i_j] > e_j)$

Algorithm:
for all combinations, c_i, of size q, $i \in \left\{1, \binom{n}{q}\right\}$, from M **do**
  for all $msg_j \in c_i, j \in \{1, q\}$ **do**
    for all $msg_k \in c_i, k \in \{i+1, q\}$ **do**
      if msg_j and msg_k are in conflict **then**
        break abort combination c_i
      end if
    end for
  end for
  $vq \leftarrow true$
  $C \leftarrow c_i$
  return vq, C
end for
$vq \leftarrow false$
$C \leftarrow \perp$
return vq, C

7 Experimental Evaluation

This chapter presents the results from a set of experiments for a single replica failure scenario. The scenario is tested using Live Replacement and Reconfiguration, together with a varying number of replicas and different sizes of replicated service state. The experimental results are used to evaluate Live Replacement and Reconfiguration with respect to the two failure handling metrics defined in Section 2.2, Delay and Disruption.

7.1 Experimental setup

7.1.1 Replicated Service

A simple key-value store, KVS, is used as the replicated service for all performed experiments. The application was developed for testing the Goxos framework together with a stateful service. The KVS application is a replicated hash map. The service replicates a map with string as the data type for both key and value. Clients can issue requests to a KVS cluster using a corresponding client application, KVSC. The operations available to clients are exactly the same as the ones available for a native Go map. These operations are read, write and delete.

CPU	Intel Xeon CPU E5606 @ 2.13GHz
Memory	16GB
Operating system	CentOS 6

Table 7.1: Specification for replica and client machines

7.1.2 Hardware

The experiments performed uses a set of identical physical machines. Both replicas and clients run on the same type of machine. The specifications for the machine type is presented in Table 7.1. The machines are connected through a switched Gigabit Ethernet local area network (LAN). A KVS replica is provided with a single individual machine. Every replica is instructed to use all available cores. This is four cores for the machines used here. Replicas connect to each other via TCP connections. A client machine host five individual clients. All clients connect to replicas using TCP connections. All machines synchronize their clocks using the Network Time Protocol (NTP) [27]. NTP can usually achieve 1 millisecond accuracy in LANs [28].

7.1.3 Failure Scenario

The scenario for the experiments performed involve failure of a single replica. The failing replica is initially part of KVS cluster which is serving a set of clients. The replica failure is handled by the application using the failure handling method specific to each experiment. The failing replica do not undertake any special role during an experiment run. This means the replica is not acting as the Paxos or Replacement leader at any time during an experiment.

The failure scenario is tested using a total of three and five replicas. The replicas are in this description denoted as n_0 to n_2 when using three replicas, and n_0 to n_4 when using five replicas. The Paxos leader is defined to be n_2 when using three replicas, and n_4 when using five replicas. The Replacement leader is defined to be n_0 when using both 3 and 5 replicas. All replicas acting as either the Paxos or Replacement leader is assumed to be stable for the whole duration of every experiment.

A crash failure is injected at replica n_1 20 seconds after starting every experiment. Every experiment ends after 60 seconds. Every client is able to finish sending their assigned number of requests before an experiment ends.

The combination of Live Replacement or Reconfiguration, three or five replicas and 1 MB or 10 MB service state, results in eight different experiments. Each individual experiment is performed 24 times. Every experiment is used to evaluate both Delay and Disruption for the specific combination of settings and employed failure handling method.

7.1.4 Experimental settings

Every client participating in an experiment only issue write requests to the replicated service. The payload for a request is 32 bytes, which includes 16 bytes for the key and 16 bytes for the value. All write requests contains a randomly chosen key from a known set of keys already present at the service. This is done to avoid changing the replicated service state during an experiment run. The corresponding request value is randomly generated at the client. Each client sends 2500 requests to the KVS cluster during a single experiment run. With a total of 50 clients and 24 runs, this results in 3 million client requests per experiment. As defined in Section 2.1, each client waits for a response to a sent request before issuing a new one. Clients issue requests at full throttle, meaning they do not wait any period of time before sending a new request after receiving a response for the previous sent.

Both replicas and clients log events using an in-memory event logger. Replicas log a specific set of events needed for measuring the Delay and Disruption observed during the experiments. Clients log the time when sending a request and when receiving a response. These times are used to calculate the request latency. The logger flushes the events to stable storage at the end of each experiment run.

The Failure Detector use for every experiment a timeout value of 1000 milliseconds. This is fairly high value, and it would in a typical LAN deployment usually be set more aggressively to achieve a lower detection time. The timeout value used for these experiments were deliberately set high. More client requests will be categorized as belonging to the period with a reduced number of replicas by increasing the duration until failure handling is triggered by the FD. The impact on latency for client requests can more accurately be evaluated with a higher amount of requests belonging to this period.

The Go garbage collector is turned off for the whole duration of all experiments. This is done for avoiding any disruption at the replicas during operation. Initial test experiments showed large regular spikes for client request latencies during runs where the garbage collector was enabled. This made it difficult to properly identify the higher request latencies observed during failure handling.

The multi-decree Paxos pipelining parameter is set to 3 for all performed experiments. The properties for all experiments as described above are summarized in Table 7.2.

7.2 Delay

This section will evaluate the experimental results with regard to Delay. Delay was defined in Section 2.2 as *the time from failure detection until a new replica is participating in Paxos*. This period consist of three individual successive failure handling

Replicated Service:	Key-Vaule Store (KVS)
Request type:	Write (Key, Value)
Request payload:	32 Bytes (Key: 16 bytes, Value: 16 bytes)
Application state:	1MB or 10MB
Pipelining parameter (α):	3
Failure handling method:	Live Replacement or Reconfiguration
Number of replicas:	3 or 5
Number of clients:	50
Number of clients per machine:	5
Number of requests sent per client:	2500
Failure detector timeout:	1000 milliseconds
Number of runs per experiment:	24

Table 7.2: Overview of experiment properties

durations:

1. **Until suspected:** The duration from a failed replica has crashed until it is suspected and failure handling is initiated.
2. **Initalization:** The duration from a new replica is started until it is initialized with an application state.
3. **Activation:** The duration from the new replica is initialized until it is participating in the Paxos protocol.

All of the three durations listed above were measured for every experiment. In addition, the duration spent by a new replica performing catch-up was measured. Duration one and two above was observed to be equal regardless of whether Live Replacement or Reconfiguration was used as failure handling method. The is expected since both methods rely on the same Goxos modules to perform their actions during theses periods. Only the activation duration is for this reason considered here for comparing the two methods. For the experiments performed the activation duration is defined for each failure handling method as follows:

- **Live Replacement:** The duration from the Replacement leader has finished initializing the replacer replica until the replacer replica has verified a valid quorum of ⟨EpochPromise⟩ messages.
- **Reconfiguration:** The duration from the Paxos leader has finished initializing the new replica until the new replica receives its first ⟨Join⟩ message.

The complete experimental data and statistics for all performed experiments can be found in Appendix A.

Section 2.2 defined the window of vulnerability (WoV) to be any period of time where a system is not operating at its initial level of fault-tolerance. This is for the scenario considered here the time interval from failure of the single replica until the new replica is activated. The activation durations measured for the experiments performed here will also be presented as a percentage of the WoV.

The measured activation durations for all experiments are shown in Table 7.3. It can be seen that the mean activation duration for Reconfiguration is stable at 5.75 to 6.77 ms for every experiment. A difference in the size of the cluster or replicated service state do not affect the activation duration for Reconfiguration to any great extent. The activation duration for Reconfiguration is a very small part of the total WoV. Increasing the replicated service state causes the WoV duration to increase since it takes a longer time to initialize a new replica. Since the activation duration remains constant for Reconfiguration, the activation duration becomes a smaller part of the WoV as the replicated service state increases.

The observations done for Reconfiguration stand in contrast to those done for Live Replacement. The mean activation durations for Live Replacement is as seen in Table 7.3 much higher for every combination of experiment variables. The activation duration for Live Replacement is approximately 30 times larger compared to Reconfiguration for the experiment using 5 replicas and 10 MB replicated service state. The activation duration for Live Replacement can also be seen to be related to the replicated service state size. A larger state size can be seen to result in a longer activation duration. A larger set of performed experiments would be needed to determine the exact relationship between these two factors.

The consistently large difference in activation durations between the two failure handling methods were naturally something that had to be investigated more closely. The main reason for the generally high Live Replacement activation durations were believed to be related to the generation of ⟨EpochPromise⟩ messages at each replica. As explained in Section 6.3.4, a replica queries the Acceptor module for state when generating an ⟨EpochPromise⟩. A replacer replica is initialized with application state corresponding to a certain slot. A replica request all acceptor state above this slot when generating an ⟨EpochPromise⟩. The amount of acceptor state embedded in the ⟨EpochPromise⟩ can become quite large if the initialization procedure takes a long time due to transferring a large amount of application state. This claim is best substantiated by presenting an example. A new replica is initialized with application state corresponding to slot x. Transferring the application state to the new replica take y seconds. During this duration the service is able to concurrently decided on client requests. The replicated service is under heavy load, and for this reason the Paxos protocol is during these y seconds able to decided on 1000 client requests, making the highest decided slot at each replica $x + 1000$. Af-

ter y seconds the Replacement leader broadcast a ⟨PREPAREEPOCH⟩ instructing the other replicas to generate an ⟨EPOCHPROMISE⟩ for the new replacer replica containing all acceptor state above slot x. Each replica as a result query their acceptor module for all acceptor state over slot x. The resulting ⟨EPOCHPROMISE⟩ contains acceptor state for at least 1000 slots. Both copying this state internally at each replica and transferring it to the new replica can take a considerable amount of time. This time is also naturally dependent on the actual size of the client requests.

An extra experiment were performed to test the hypothesis described above. The results from this experiment is presented in Section 7.4.

The increased activation durations observed for Live Replacement also has a direct impact on the time spent doing catch-up for a new replica. Catch-up is in this context the procedure performed by a new replica for obtaining any slots decided concurrently by the other replicas during its initialization and activation. More decided slots may need to be transfered when the activation duration increase. It can be seen from the data in Appendix A that the experiments using Live Replacement generally has a higher catch-up duration compared to Reconfiguration.

7.3 Disruption

This discussion will now evaluate the experimental results with respect to Disruption. Disruption was defined in Section 2.2 as *the additional latency state machine requests experience during failure handling*. Disruption are for the experiments performed here measured from the clients point of view. This is done by measuring the round trip time for every client request sent. Round trip time (RTT) denotes the duration from a client sends a request to when it receives a response for the request. Every measured client request RTT were categorized into one of the failure handling periods explained in Section 7.2 or categorized as belonging to the period with no failure handling performed. The complete experimental data for measured client request latencies can be found in Appendix A.

An example of two typical client runs will be presented before discussing the general observations. Figure 7.1 show the RTTs for a single client. The data is taken from a randomly chosen run from the experiment using Live Replacement, three replicas and 1 MB application state. The set of events related to failure handling at the replicas is also marked in the figure. A spike in the request latency can be seen directly after the failed replica is suspected. This is due to the Replacement leader initializing the new replica. The Replacement leader is while initializing the new replica also participating in Paxos. The quorum size is two when using three replicas, and the Paxos and Replacement leader are the only two operational replicas after the failure. Both replicas must participate to form a quorum. Additional latency is experienced since the Replacement leader execute the Paxos protocol slower due

3 replicas, 1 MB replicated service state		
	Live Replacement	Reconfiguration
Activation duration mean	70.1 ms	5.75 ms
Activation duration *SD*	11.5 ms	2.66
Mean activation duration as % of WoV	5.39%	0.45%
3 replicas, 10 MB replicated service state		
	Live Replacement	Reconfiguration
Activation duration mean	160 ms	5.85 ms
Activation duration *SD*	17.0 ms	2.77 ms
Mean activation duration as % of WoV	4.41 %	0.17 %
5 replicas, 1 MB replicated service state		
	Live Replacement	Reconfiguration
Activation duration mean	56.2 ms	6.62 ms
Activation duration *SD*	24.3 ms	2.65 ms
Mean activation duration as % of WoV	4.28%	0.51%
5 replicas, 10 MB replicated service state		
	Live Replacement	Reconfiguration
Activation duration mean	206 ms	6.77 ms
Activation duration *SD*	72.0 ms	3.10 ms
Mean activation duration as % of WoV	5.51 %	0.19%

Table 7.3: Activation duration overview

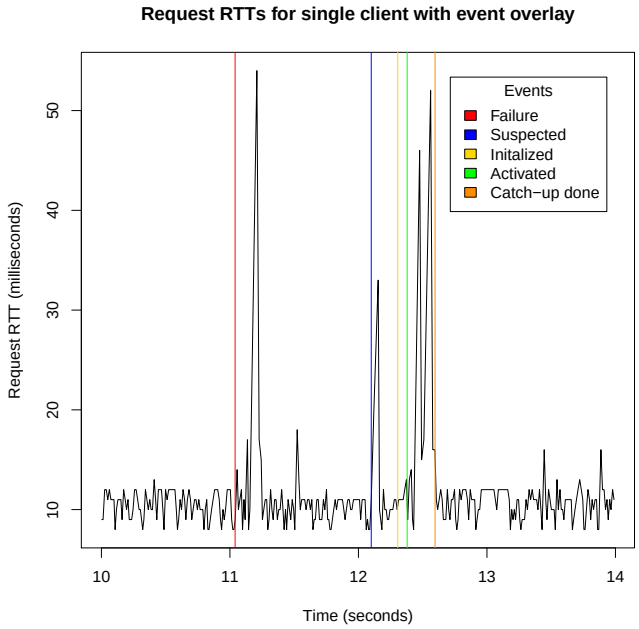

Figure 7.1: Request RTTs for a single client with event overlay, Live Replacement, 3 replicas, 1 MB replicated service state

to the concurrent initialization of the new replica.

Two large spikes can be seen during the catch-up period for the new replica. A new replica queries the Paxos leader during catch-up. This naturally disrupts the Paxos leader from making proposals and can result in increased latencies for every request sent from any client during this period. The Goxos implementation should in the future consider adjusting what replica that is contacted during catch-up for avoiding disruption of the Paxos leader. A randomized replica selection scheme could for example be applied.

Figure 7.2 show the RTTs for the requests sent by a single client for the same experiment settings as described above, but here using Reconfiguration. The activation duration is hard to identify in the figure due to the small time interval. Only one latency spike is observed for the catch-up interval. This may be due to the reduced duration of catch-up observed when using Reconfiguration. Both figures described above show an increase in latency for a small duration after the replica has failed. This spike is observed regardless of failure handling method. The information logged during the experiments were not sufficient for accurately identifying the reason for these spikes.

The measured mean client request RTT during activation is presented in Table 7.4. As for the activation durations, Reconfiguration also here has relatively

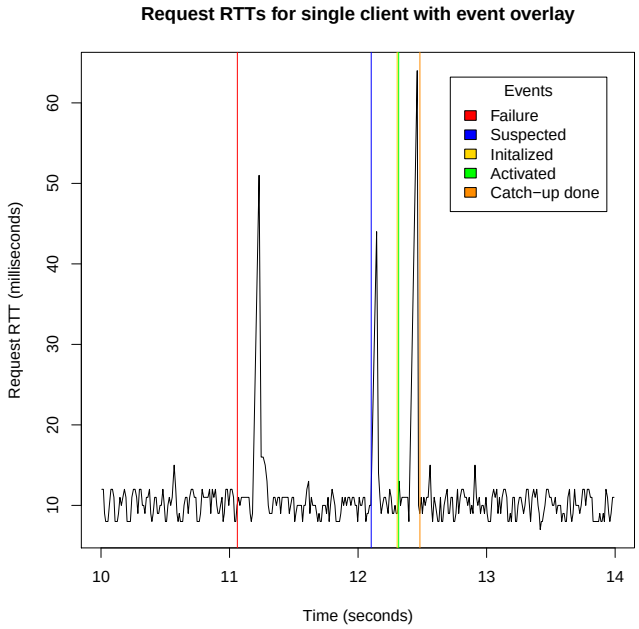

Figure 7.2: Request RTTs for a single client with event overlay, Reconfiguration, 3 replicas, 1 MB replicated service state

stable values regardless of cluster and replicated service state size. Live Replacement has a lower mean RTT value for the experiment using three replicas and 1 MB state. The method has for the other experiments a much higher RTT value than Reconfiguration. Live Replacement has in addition a generally higher standard deviation. The experiment using five replicas and 10 MB state has with Live Replacement a very high standard deviation. It can be seen from Table A.8 that median request RTT for this experiment is 11 ms. This value could serve as a better indication of expected RTT since the mean is affected by outliers. The presences of these outliers still indicate more unpredictable RTTs during activation when using Live Replacement compared to Reconfiguration.

The increased Disruption measured for Live Replacement during the activation interval is again believed to be related to the ⟨EpochPromise⟩ generation at each replica. Creating and sending this message can as argued in Section 7.2 take a considerable amount of time due the replicated service state size and a high workload for clients. The ⟨EpochPromise⟩ is generated and sent by every replica. The Acceptor module is blocked at each replica when creating this message. As a result the entire cluster is disrupted for the time period needed to generate and send the ⟨EpochPromise⟩. This will naturally be visible to clients through increased request RTTs. As stated in Section 7.2, an extra experiment was performed to try to

3 replicas, 1 MB replicated service state		
	Live Replacement	Reconfiguration
Client request RTT mean during activation	11.07 ms	11.86 ms
Client request RTT *SD* during activation	2.40 ms	1.48 ms
3 replicas, 10 MB replicated service state		
	Live Replacement	Reconfiguration
Client request RTT mean during activation	75.04 ms	12.77 ms
Client request RTT *SD* during activation	23.83 ms	6.25 ms
5 replicas, 1 MB replicated service state		
	Live Replacement	Reconfiguration
Client request RTT mean during activation	23.18 ms	12.51 ms
Client request RTT *SD* during activation	43.98 ms	2.11 ms
5 replicas, 10 MB replicated service state		
	Live Replacement	Reconfiguration
Client request RTT mean during activation	75.04 ms	14.01 ms
Client request RTT *SD* during activation	105.5 ms	9.08 ms

Table 7.4: Overview of client request RTT during activation

verify this assumption.

It is finally noted that the implementation of Live Replacement add a small performance overhead to every request decided at the replicated service. There are two reasons for this. The ⟨LEARN⟩ messages sent over the network are larger in size because they contain a replica's epoch vector. Secondly, a simple valid quorum check is performed for every ⟨LEARN⟩ used as part of a quorum. This performance overhead increase with the number of replicas used. This mean that the measured client request RTTs for Live Replacement and Reconfiguration are not directly comparable. However, the experiments performed here has shown the performance impact on Live Replacement to be very small. From the data presented in Appendix A, the difference can be seen to be in the range of 0.2 to 0.5 ms, depending on the number of replicas.

7.4 Live Replacement Adjusted

This section present an additional experiment performed to test the Live Replacement implementation with an adjustment to the ⟨EPOCHPROMISE⟩ generation.

The motivation for the experiment was to verify if the current ⟨EpochPromise⟩ generation method were the main reason for the high Delay and Disruption observed for Live Replacement in the experiments discussed in Section 7.2 and Section 7.3.

The ⟨EpochPromise⟩ generation were adjusted on one single point. The initial implementation queries the Acceptor module for all its state above the slot that correspond to the application state at the new initialized replica. The adjusted implementation instruct each replica to only query the Acceptor module for any state above the highest slot it has executed. This mean only acceptor state for any concurrently running instances is sent with the ⟨EpochPromise⟩. This state is limited by the pipelining paramenter (α). The adjustment to Live Replacement do not violate *saftey* of the Paxos State Machine, but may change the conditions for when *liveness* is guaranteed. The original implementation can guarantee liveness as long as no more than f acceptors fail before a replacement is complete. The liveness guarantees for the adjusted version of Live Replacement has not been clearly identified. The precise theoretical implications of the discussed protocol adjustment should be investigated in more detail.

The adjusted implementation was tested using five replicas and 10 MB application state. The experimental results are presented in Table 7.5. The table contains both mean activation duration and client request RTT measurements. Data from the corresponding experiments using Reconfiguration and the original Live Replacement implementation is also included for comparison. It can be seen that the adjusted Live Replacement implementation has a mean activation duration of 15.1 ms. This is much lower than the 206 ms observed as the mean for the original implementation. The standard deviation is also much lower for the adjusted implementation. The relative standard deviation is approximately 35.0% for the original implementation and 11.4% for the adjusted. The mean activation duration for the adjusted Live Replacement implementation is also for this experiment higher than the mean duration observed for Reconfiguration. Figure 7.3 shows the calculated density estimates for the activation durations discussed here. Kernel density estimation is a non-parametric way to estimate the probability density function of a random variable [29]. The estimates were obtained using a Gaussian kernel. The estimates reflect the points made in the discussion above. Reconfiguration has a lower expected activation duration than Live Replacement. The larger variation in Reconfiguration measurements is also reflected in the estimate.

The mean client request RTT for the adjusted version of Live Replacement is also much lower than for the original implementation. The mean RTT value can be seen to have dropped from 75.04 to 14.31 ms. The mean RTT value for the adjusted Live Replacement version is basically the same as the value observed for Reconfiguration. The difference of 0.3 ms is approximately the difference that can be expected when taking into account the discussed overhead of Live Replacement. A density

5 replicas, 10 MB replicated service state			
	Live Replacement	Live Replacement*	Reconfiguration
Activation duration mean	206 ms	15.1 ms	6.77 ms
Activation duration *SD*	72.0 ms	1.72 ms	3.10 ms
Act. dur. mean as % of WoV	5.51%	0.30%	0.19%
Client request RTT mean	75.04 ms	14.31 ms	14.01 ms
Client request RTT *SD*	105.5 ms	11.61 ms	9.08 ms

Table 7.5: Comparison of Live Replacement, Live Replacement* and Reconfiguration

estimate for the client request RTT during activation can be seen in Figure 7.4. The estimates were obtained using a Gaussian kernel. It can be seen that estimates for both methods are affected by outliers in the range from 40 to 65 ms.

The observations from the additional experiment clearly identify the importance of how ⟨EpochPromise⟩ messages are generated at each replica. It is clear that the approach taken in the initial implementation is not a viable option when comparing Live Replacement to Reconfiguration. The results show a high amount of both Delay and Disruption for Live Replacement when using this approach. The adjusted version of Live Replacement is observed to perform much better. The version is competitive against Reconfiguration with respect to Disruption. The adjusted Live Replacement version is observed to have slightly higher Delay compared to Reconfiguration. All experiments were performed with $\alpha = 3$. This parameter is important for the waiting variant of Reconfiguration implemented for this thesis. Further experiments should be performed to investigate the effect of adjusting the pipelining parameter.

Reconfiguration performed well for all experiments. The Reconfiguration implementation mainly use the existing Paxos modules and messages when performing failure handling. The Paxos modules have been extensively tested since they are a part of the core Goxos framework. This fact may contribute to the observed robustness of Reconfiguration. The Live Replacement implementation use a separate module for handling and sending ⟨PrepareEpoch⟩ and ⟨EpochPromise⟩ messages. Using a separate module may lead to increased Delay since both the Replacement handler and Paxos modules operate concurrently during failure handling. The prototype implementation of Live Replacement is also based on a newly presented theoretical description. It may be unrealistic to expect that a first attempt on implementing a new protocol would result in optimal performance. Such an initial implementation need more testing and adjustments before performing optimally.

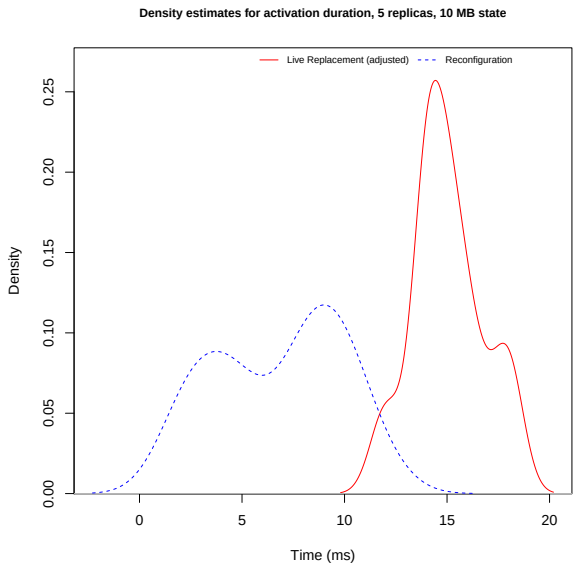

Figure 7.3: Density estimates for activation duration for Live Replacement (adjusted) and Reconfiguration

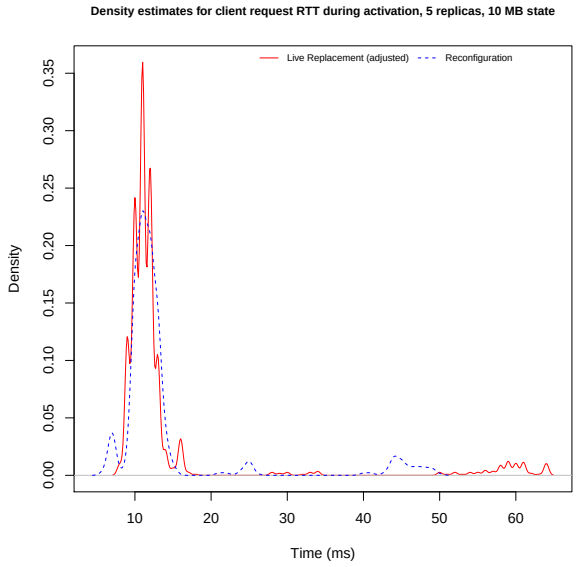

Figure 7.4: Density estimates for client request RTT Live Replacement (adjusted) and Reconfiguration during activation

8 Conclusion and Further Work

8.1 Conclusion

This thesis presents an implementation and experimental evaluation of two methods for failure handling in Paxos State Machines: Live Replacement [3] and Reconfiguration [5]. The two methods can be applied to replicated state machines (RSMs) [1] based on the Paxos protocol [2]. The two failure handling methods considered for this thesis is evaluated in the context of immediate failure handling. Such failure handling methods should enable an RSM to restore its initial fault tolerance as soon as possible. The methods should in addition to being fast also have a minimal impact on the performance of an RSM. This mean they should degrade the state machine throughput as little as possible during failure handling. Immediate failure handling is a central component for enabling replicated services to maintain high availability [3, 7].

The work on this thesis started with the design and implementation of Live Replacement. The failure handling method were developed as a part of the Goxos [4] framework. Live Replacement is a new method for immediate failure handling. It differs from more classical failure handling methods since the method operates independent of state machine progress. Live Replacement was at the beginning of this work available as theoretical description. The design and implementation work therefore had to address several design and implementation details not addressed in the original description.

Another goal for this thesis was to evaluate Live Replacement against a set of

metrics defined for immediate failure handling and against other methods relevant for immediate failure handling. For this reason a was Reconfiguration [5], a traditional and well known mechanism for failure handling, implemented as part of the Goxos framework.

The experimental evaluation performed as part of this thesis show that the initial implementation of Live Replacement need further testing and adjustments before performing up to par with Reconfiguration for a basic failure scenario. Further experiments should be conducted to evaluate the protocol for more complex failure scenarios. The evaluation also shed light on a subtle but important implementation detail for Live Replacement that greatly affected the performance of the method. It was identified that choices made when implementing certain protocol steps of the Live Replacement method can have a very large impact on both Delay and Disruption.

8.2 Further Work

This is thesis is concluded with a discussion concerning possible optimizations and further work.

Further Experiments The evaluation conducted as a part of this thesis use a basic single failure scenario for the experiments performed. More complex failure scenarios should be tested in experiments for evaluating the presented implementation further. The experiments performed for this thesis does also not include scenarios involving network partitions. This is due to the increased complexity of simulating network partitions versus simpler crash failures. A relevant scenario related to Live Replacement would be to measure the performance impact of when nodes unaware of that they have been replaced. Such scenarios could arise due to network partitions, and could trigger the other replicas to enable strict valid quorum verification for several slots. Such behavior could impair throughput and increase Disruption until a replaced node is aware of its replacement and terminates.

The current implementation of Live Replacement does at the time being not allow concurrent replacements. It would naturally be interesting to allow such behavior, and in relation perform experiments involving failures in rapid succession. Such scenarios should in at least in theory prove particularly advantageous for Live Replacement compared against Reconfiguration. This is assuming that the Reconfiguration algorithm reacts instantly to the first failure experienced, and does not wait for a certain number of failures before proposing the reconfiguration command.

Another interesting experiment could involve running a set of replicas using autonomous failure handling, and measure the duration until the RSM experiences a critical scenario where it is unable repair itself and make progress. Failures would have to be injected at each replica with certain independent probability.

Improved Failure Information All failure handling in the Goxos framework is initiated by the implemented failure detector. The failure detector used in this thesis is as described in Section 6.1.1 implemented as an *eventually perfect failure detector*. The failure detector encapsulates the timing assumptions of a partially synchronous system. When using the failure detector to trigger all failure handling, the provided information is naturally wanted be as accurate as possible. The failure detector considered here can by definition provide false suspicions during periods of asynchrony in the system. Reacting to such false suspicions may cause a costly failure handling operation to start, even if the system for example only experienced a short transient network partition.

A new abstraction called *failure informer* is presented in [30]. This abstraction could to a certain extent help avoid such unnecessary failure handling. The abstraction and its corresponding implementation allows applications to take application-specific recovery actions. It reports a small number of conditions that each represent a class of problems that affect the applications. These conditions are also reported with different levels of accuracy, ranging from certain occurrence and certain permanence to certain occurrence and uncertain permanence. Uncertain reports are also helpful to systems, as they can consider the cost-benefit trade-offs of waiting versus initiating failure handling for both reported problems of uncertain permanence or occurrence.

The presented failure reporting service implementing this abstraction is restricted to a single administrative domain. This is not an unreasonable assumption is in the context of this thesis, where at least the RSM cluster can be viewed as running in a single data center or enterprise network. Another interesting feature of the failure reporting service is the ability to also provide warnings about possible eminent failures. Examples of such failures could be CPU overload at a replica or a saturation of a network link. Access to such information could be for very beneficial for Live Replacement (and Reconfiguration), as it enables preemptive action with the goal of reducing the window of vulnerability for keeping the fault-tolerance level of the system high.

Application State Transfer Application state transfer is for the two faliure handling techniques relevant for this thesis clearly the of the operation with the highest impact on Delay if the state is large. A interesting approach to the state transfer challenge could be to use additional replicas as hot standbys. Such replicas would operate in an special standby mode. They would only be listening for

messages containing decided requests from replicas participating in the RSM cluster. Using Live Replacement, such a hot standby could be switched into operation very quickly. This would involve practically no state transfer, reducing Delay to a minimum. This approach comes at the cost of constantly sending messages with decided requests to hot-spares, as well as the computing resources needed for having such standbys operational.

Implementation Details In its current form, Live Replacement and its corresponding adjusted version of multi-decree Paxos, performs a simple valid quorum check for each received PROMISE and LEARN message. This procedure, as mentioned in Section 7.3, results in a small performance penalty for the Paxos protocol. It should be possible to adjust the implementation so that this check is only performed in periods where replacements are in progress. The needed logic for keeping track of the related events are not yet implemented.

Another possible implementation optimization is related to how a replica obtain an application state snapshot in conjunction with initiating failure handling. The request from the replica to the service layer is in the current implementation a blocking method call. This request can be costly in relation to state machine progress if the service state is large and needs to be properly serialized. This is because a replica is prevented from executing decided request during this operation. Special precautions need to be taken if a replica should be able to continue executing decided request while obtaining a snapshot. The technique mentioned in [15] uses a shadow data structure to track updates while the underlying application state is serialized. Such an improvement would be especially relevant for the Reconfiguration technique because the replica acting as Paxos leader would to a certain degree be able to handle decided requests while concurrently requesting for an application state snapshot. As a consequence Disruption could be reduced.

A

Complete Experimental Data

This appendix contains the complete experimental data obtained from the nine experiments presented in Chapter 7. The data is presented using two tables for each experiment, one for failure handling durations and one for client request RTT statistics. The tables are grouped as corresponding Live Replacement and Reconfiguration pairs, with the exception of Experiment 9.

Failure handling durations	Mean	*SD*	*SE*	Min	Max	Median
Until suspected	1032	16.95	3.459	1004	1064	1027
Initialization	198.2	4.053	0.827	193	206	197
Activation	70.12	11.46	2.340	17	78	72.5
Catch-up	206.6	16.14	3.293	153	235	205.5
WoV	1302	20.19	4.121	1250	1341	1300
WoV & catch-up	1509	30.87	6.302	1404	1555	1511

Table A.1: Failure handling durations: Exp. 1 - Live Replacement, 3 replicas and 1 MB state. Unit ms.

Failure handling durations	Mean	*SD*	*SE*	Min	Max	Median
Until suspected	1059	23.82	4.863	1012	1111	1053
Initialization	198.9	3.888	0.793	192	210	198.5
Activation	5.75	2.66	0.54	2	11	6
Catch-up	159.8	6.257	1.277	149	173	159
WoV	1277	24.21	4.941	1217	1313	1260
WoV & catch-up	1425	26.59	5.427	1370	1483	1424

Table A.2: Failure handling durations: Exp. 2 - Reconfiguration, 3 replicas and 1 MB state. Unit ms.

Client request RTT	Mean	*SD*	*SE*	Min	Max	Median	% of total number of requests
Outside WoV & catch-up	10.34	1.52	0.0009	1	36	10	95.14
Before initialization	11.44	9.17	0.0285	1	163	10	3.44
During initialization	12.15	6.40	0.0471	4	55	11	0.62
During activation	11.07	2.40	0.0281	7	21	11	0.24
During catch-up	14.26	14.31	0.1103	6	13	11	0.56

Table A.3: Client request RTT: Exp. 1 - Live Replacement, 3 replicas and 1 MB state. Unit ms.

Client request RTT	Mean	*SD*	*SE*	Min	Max	Median	% of total number of requests
Outside WoV & catch-up	9.995	1.899	0.0011	1	37	10	95.42
Before initialization	11.58	9.521	0.0292	3	163	10	3.53
During initialization	10.10	2.593	0.0191	4	50	10	0.61
During activation	11.86	1.481	0.0573	8	18	12	0.02
During catch-up	14.91	13.54	0.1227	4	76	11	0.41

Table A.4: Client request RTT: Exp. 2 - Reconfiguration, 3 replicas and 1 MB state. Unit ms.

Failure handling durations	Mean	*SD*	*SE*	Min	Max	Median
Until suspected	1556	23.57	4.811	1522	1604	1549
Initialization	1913	6.022	1.229	1904	1927	1912
Activation	160	17.0	3.47	113	173	167
Catch-up	1344	25.52	5.209	1280	1380	1347
WoV	3629	26.57	5.424	3592	3687	3622
WoV & catch-up	4974	46.50	9.491	4875	5067	4970

Table A.5: Failure handling durations: Exp. 3 - Live Replacement, 3 replicas and 10 MB state. Unit ms.

Failure handling durations	Mean	*SD*	*SE*	Min	Max	Median
Until suspected	1580	24.98	4.806	1538	1636	1578
Initialization	1911	4.414	0.849	1903	1922	1911
Activation	5.852	2.769	0.533	2	13	5
Catch-up	1292	23.53	4.526	1259	1336	1294
WoV	3499	25.84	4.974	3454	3563	3496
WoV & catch-up	4792	40.54	7.802	4712	4880	4798

Table A.6: Failure handling durations: Exp. 4 - Reconfiguration, 3 replicas and 10 MB state. Unit ms.

Client request RTT	Mean	*SD*	*SE*	Min	Max	Median	% of total number of requests
Outside WoV & catch-up	10.07	1.907	0.0009	1	45	10	88.14
Before initialization	13.36	25.87	0.0285	3	964	10	4.51
During initialization	12.78	28.43	0.0471	3	390	10	5.51
During activation	18.75	23.83	0.0281	1	121	10	0.31
During catch-up	34.63	147.5	0.1103	1	1062	11	1.53

Table A.7: Client request RTT: Exp. 3 - Live Replacement, 3 replicas and 10 MB state. Unit ms.

Client request RTT	Mean	*SD*	*SE*	Min	Max	Median	% of total number of requests
Outside WoV & catch-up	10.25	1.602	0.0009	1	370	10	89.10
Before initialization	16.18	36.24	0.0921	3	378	11	4.58
During initialization	10.79	9.084	0.0210	5	374	10	5.53
During activation	12.77	6.247	0.2270	8	50	12	0.02
During catch-up	69.61	231.6	1.445	6	1425	11	0.76

Table A.8: Client request RTT: Exp. 4 - Reconfiguration, 3 replicas and 10 MB state. Unit ms.

Failure handling durations	Mean	*SD*	*SE*	Min	Max	Median
Until suspected	1055	38.84	7.767	1002	1123	1052
Initialization	200.3	4.279	0.853	192.0	211.0	199.0
Activation	56.24	24.26	4.851	22.00	122.0	70.00
Catch-up	227.4	66.86	13.37	166.0	395.0	198.0
WoV	1313	44.31	8.862	1251	1393	1303
WoV & catch-up	1541	73.62	14.72	1447	1731	1541

Table A.9: Failure handling durations: Exp. 5 - Live Replacement, 5 replicas and 1 MB state. Unit ms.

Failure handling durations	Mean	*SD*	*SE*	Min	Max	Median
Until suspected	1087	37.34	7.323	1022	1196	1080
Initialization	202.4	4.089	0.802	195	209	202
Activation	6.615	2.654	0.521	3	12	5.5
Catch-up	158	9.506	1.864	143	178	157
WoV	1297	37.72	7.398	1234	1406	1292
WoV & catch-up	1456	38.19	7.490	1393	1560	1450

Table A.10: Failure handling durations: Exp. 6 - Reconfiguration, 5 replicas and 1 MB state. Unit ms.

Client request RTT	Mean	*SD*	*SE*	Min	Max	Median	% of total number of requests
Outside WoV & catch-up	10.67	4.023	0.0023	1	736	10	95.55
Before initialization	12.11	17.47	0.0548	6	661	10	3.24
During initialization	11.35	5.474	0.0376	7	226	10	0.68
During activation	23.18	43.98	0.6471	8	239	12	0.15
During catch-up	20.65	26.75	0.2472	7	158	11	0.37

Table A.11: Client request RTT: Exp. 5 - Live Replacement, 5 replicas and 1 MB state. Unit ms.

Client request RTT	Mean	*SD*	*SE*	Min	Max	Median	% of total number of requests
Outside WoV & catch-up	10.35	2.434	0.0013	1	122	10	95.39
Before initialization	11.54	8.441	0.0246	3	145	10	3.63
During initialization	10.67	3.689	0.0265	2	54	10	0.60
During activation	12.51	2.106	0.0746	4	18	12	0.02
During catch-up	16.01	14.94	0.1366	1	98	11	0.37

Table A.12: Client request RTT: Exp. 6 - Reconfiguration, 5 replicas and 1 MB state. Unit ms.

Failure handling durations	Mean	*SD*	*SE*	Min	Max	Median
Until suspected	1599	74.03	13.52	1528	1793	1572
Initialization	1918	16.68	3.045	1910	1959	1923
Activation	205.8	72.04	13.15	150	361	168
Catch-up	1572	92.17	16.83	1327	1727	1575
WoV	3737	96.38	17.57	3623	3900	3715
WoV & catch-up	5310	162.6	29.69	4958	5590	5314

Table A.13: Failure handling durations: Exp. 7 - Live Replacement, 5 replicas and 10 MB state. Unit ms.

Failure handling durations	Mean	*SD*	*SE*	Min	Max	Median
Until suspected	1598	23.61	4.462	1570	1671	1592
Initialization	1919	6.475	1.227	1910	1935	1918
Activation	6.768	3.096	0.585	2	12	8
Catch-up	1270	24.99	4.723	1213	1324	1267
WoV	3525	26.17	4.946	3484	3597	3520
WoV & catch-up	4796	33.97	6.420	4744	4860	4786

Table A.14: Failure handling durations: Exp. 8 - Reconfiguration, 5 replicas and 10 MB state. Unit ms.

Client request RTT	Mean	*SD*	*SE*	Min	Max	Median	% of total number of requests
Outside WoV & catch-up	10.62	3.619	0.0020	1	157	10	88.09
Before initialization	14.47	27.91	0.0698	3	878	10	4.26
During initialization	11.50	11.81	0.0241	5	414	10	6.41
During activation	75.04	105.5	1.467	8	343	11	0.14
During catch-up	56.96	225.6	1.108	6	1368	11	1.11

Table A.15: Client request RTT: Exp. 7 - Live Replacement, 5 replicas and 10 MB state. Unit ms.

Client request RTT	Mean	*SD*	*SE*	Min	Max	Median	% of total number of requests
Outside WoV & catch-up	10.17	2.455	0.0013	1	120	10	89.25
Before initialization	15.42	33.77	0.0836	2	389	10	4.67
During initialization	11.07	12.45	0.0287	3	390	10	5.38
During activation	14.01	9.081	0.3097	6	50	12	0.03
During catch-up	73.37	219.8	1.423	4	1056	11	0.68

Table A.16: Client request RTT: Exp. 8 - Reconfiguration, 5 replicas and 10 MB state. Unit ms.

Failure handling durations	Mean	*SD*	*SE*	Min	Max	Median
Until suspected	1585	57.10	12.46	1472	1702	1582
Initialization	1924	16.93	3.695	1910	1959	1924
Activation	15.05	1.717	0.3746	12	18	15
Catch-up	1549	99.02	21.61	1437	1858	1523
WoV	3532	61.59	13.44	3437	3676	3521
WoV & catch-up	5082	95.02	20.73	4978	5296	5043

Table A.17: Failure handling durations: Exp. 9 - Live Replacement variation, 5 replicas and 10 MB state. Unit ms.

Client request RTT	Mean	*SD*	*SE*	Min	Max	Median	% of total number of requests
Outside WoV & catch-up	10.59	2.997	0.0019	1	146	10	89.69
Before initialization	14.21	25.91	0.0771	6	980	10	4.30
During initialization	11.40	7.910	0.0192	5	179	10	6.47
During activation	14.31	11.61	0.3061	8	64	11	0.05
During catch-up	128.4	339.1	2.998	8	1623	13	0.49

Table A.18: Client request RTT: Exp. 9 - Live Replacement variation, 5 replicas and 10 MB state. Unit ms.

Bibliography

[1] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[2] Leslie Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, December 2001.

[3] Leander Jehl and Hein Meling. Live replacement: Fast and Efficient Failure Handling for Paxos State Machines. Submitted for publication May 2013.

[4] Stephen Michael Jothen and Tormod Erevik Lea. Goxos: A Paxos Implementation in the Go Programming Language, 2013.

[5] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

[6] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2nd edition. edition, 2 2011.

[7] James W. Anderson, Hein Meling, Alexander Rasmussen, Amin Vahdat, and Keith Marzullo. Local recovery for high availability in strongly consistent cloud services. Submitted for publication, 2013.

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '83, pages 1–7, New York, NY, USA, 1983. ACM.

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

[10] Nuno Santos and André Schiper. Tuning paxos for high-throughput with batching and pipelining. In *Proceedings of the 13th international conference*

on Distributed Computing and Networking, ICDCN'12, pages 153–167, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] The Go Project. The Go Programming Language. `http://www.golang.org`, 2013. [Online; accessed 01-May-2013].

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[13] The Go Authors. Effective Go. `http://golang.org/doc/effective_go.html`, 2013. [Online; accessed 26-May-2013].

[14] The Go Authors. The Go Programming Language Specification. `http://golang.org/ref/spec`, 2013. [Online; accessed 26-May-2013].

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

[16] Stephen M. Jothen. Acropolis: Aggregated Client Request Ordering by Paxos. Master's thesis, University of Stavanger, 2013.

[17] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[18] Keith Rarick. Introducing Doozer. `http://doozer-sdec2011.heroku.com/`, 2011. [Online; accessed 01-Des-2012].

[19] Jan Kończak, Nuno Santos, Tomasz Zurkowski, Paweł T. Wojciechowski, and André Schiper. JPaxos: State machine replication based on the Paxos protocol. Technical Report EPFL-REPORT-167765, Faculté Informatique et Communications, EPFL, July 2011. 38pp.

[20] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[21] J. Kirsch and Y. Amir. Paxos for System Builders. Technical Report CNDS-2008-2, Johns Hopkins University, March 2008. 35pp.

[22] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 103–115, New York, NY, USA, 2006. ACM.

[23] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[24] Steven Pigeon. Pairing Function - From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein. `http://mathworld.wolfram.com/PairingFunction.html`, 2013. [Online; accessed 09-May-2013].

[25] Wikipedia. Pairing function — Wikipedia, the free encyclopedia, 2013. [Online; accessed 09-May-2013].

[26] Mark Summerfield. *Programming in Go: Creating Applications for the 21st Century (Developer's Library)*. Addison-Wesley Professional, 1 edition, 5 2012.

[27] David L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991.

[28] Wikipedia. Network time protocol — Wikipedia, the free encyclopedia, 2013. [Online; accessed 10-June-2013].

[29] Wikipedia. Kernel density estimation — Wikipedia, the free encyclopedia, 2013. [Online; accessed 10-June-2013].

[30] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *NSDI*, 2013.