MASTER'S THESIS 2024

Navigating Failures in Distributed Systems: A Comparative Study of Failure Detection Algorithms

Hannes Brinklert, Johan Åkerman

ISSN 1650-2884
LU-CS-EX: 2024-20

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-20

Navigating Failures in Distributed Systems: A Comparative Study of Failure Detection Algorithms

Navigera felaktigheter hos distribuerade system: en jämförande studie av feldetekteringsalgoritmer

Hannes Brinklert, Johan Åkerman

Navigating Failures in Distributed Systems: A Comparative Study of Failure Detection Algorithms

Hannes Brinklert
ha7075br-s@student.lu.se

Johan Åkerman
jo3460ak-s@student.lu.se

May 21, 2024

Master's thesis work carried out at Neo4j Sweden AB.

Supervisors: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Aleksey Karasavov, aleksey.karasavov@neo4j.com

Examiner: Michael Doggett, michael.doggett@cs.lth.se

Abstract

Failure detection algorithms are used to identify unhealthy nodes in distributed systems. The goal of this study was to improve Neo4j's use of failure detection algorithms by exploring two paths: either optimising their existing Lighthouse algorithm or by implementing a new algorithm.

Existing algorithms were surveyed and the SWIM algorithm was implemented. A baseline was established and evaluated against parameter-optimised versions of SWIM and Lighthouse in a simulated network. The results show that Baseline is scalable and reliable but slow, Lighthouse is fast but less accurate, and SWIM is moderately fast and the least accurate but generates the least network load.

In conclusion, the chosen parameters of a failure detector are to a great extent more important than the algorithm itself. Furthermore, to successfully optimise parameters it is crucial to have a scalable simulator and precise system requirements to manage the trade-off between speed, accuracy, and network load.

Keywords: Distributed Systems, Graph Databases, Failure Detection Algorithm, Membership Protocol, SWIM

Acknowledgements

After five years of studies in Lund, we would like to thank all of the people that we have met during these years who have made it an unforgettable time. We would like to especially thank our classmates for all the fun days spent at Studiecentrum filled with studying, laughter, and walks around the lake.

Further on, we would like to direct a thank you to Jonas Skeppstedt, our thesis supervisor at Lund University, for all of your valuable advice and interesting classes.

Lastly, we would like to thank everyone at Neo4j and our supervisor Aleksey Karasavov in particular. Thank you for your enthusiasm for engineering and hours of discussion. Without you, this thesis would be impossible.

Contents

1 Introduction **7**
- 1.1 Research Goals and Project Scope . . . 7
- 1.2 Related Work . . . 8
- 1.3 Scientific Contribution . . . 10
- 1.4 Disposition . . . 10
- 1.5 Contribution Statement . . . 11

2 Theory **13**
- 2.1 Graph Databases . . . 13
- 2.2 Distributed Systems . . . 14
- 2.3 Failure Detection . . . 16
  - 2.3.1 Failures . . . 16
  - 2.3.2 Key Properties and Classes . . . 16
  - 2.3.3 Evaluation Criteria . . . 17
  - 2.3.4 Algorithms . . . 17
  - 2.3.5 SWIM . . . 18
  - 2.3.6 Lighthouse . . . 19

3 Method **21**
- 3.1 Phase 1: Research . . . 21
- 3.2 Phase 2: Implementation . . . 21
- 3.3 Phase 3: Evaluation . . . 22
  - 3.3.1 Evaluation Criteria . . . 22
  - 3.3.2 Experimental Setup . . . 22
  - 3.3.3 Establishing a Baseline . . . 23
  - 3.3.4 Parameter Tuning . . . 23
  - 3.3.5 Evaluating Optimal Configurations . . . 24

4 Implementation **25**
- 4.1 Lighthouse . . . 25

4.2 SWIM . . . 27
4.2.1 Outgoing Traffic . . . 27
4.2.2 Incoming Traffic . . . 28
4.2.3 Membership List . . . 28
4.2.4 Configuration and Building . . . 29
4.2.5 Verification of SWIM . . . 29
4.3 Testbed . . . 30
4.3.1 The Original Testbed . . . 30
4.3.2 The Modified Testbed . . . 31

5 Results **37**
5.1 Parameter Optimisation . . . 37
5.2 Further Investigations . . . 41
5.2.1 Link Failure . . . 41
5.2.2 Scalability . . . 42

6 Discussion **45**
6.1 Analysis of Results . . . 45
6.1.1 Parameter Optimisation . . . 45
6.1.2 Variation . . . 46
6.1.3 Link Failure . . . 46
6.1.4 Scalability . . . 46
6.2 Limitations and Threats to Validity . . . 48

7 Conclusion **49**
7.1 Research Questions . . . 49
7.2 Future Work . . . 50

References **51**

Chapter 1
Introduction

Internet-based services are today a vital part of society as they provide the tools, services, and infrastructure that we have grown accustomed to. In 2024, Forbes reported that over 5.35 billion people around the world are frequent internet users as they spend over six hours of their days connected to the internet [1]. Hence, it is now more important than ever for digital services to be fast, reliable, and scalable to handle the surging demand.

Distributed systems are commonly used to host more demanding applications as they divide the computing power across a set of distributed machines instead of relying on a single machine [2]. However, managing distributed systems faces a set of engineering challenges that grows in complexity as the number of machines increases. One of the main challenges is coordination as each machine has its own perception of the system's status as a whole [3]. To tackle this challenge, failure detection algorithms are crucial as they identify and communicate failures, e.g. that a machine has crashed.

This thesis work was carried out at Neo4j, a world leader within the field of graph databases and whose software is trusted by 75% of the Fortune 100 companies [4]. Neo4j leverages distributed systems and is currently detecting failures by using the Akka library. However, due to increasingly expensive licensing costs, lack of customization, and troubleshooting issues, Neo4j is currently developing an in-house replacement for Akka called Lighthouse. Even though Lighthouse already has the functionality to detect failures, Neo4j wishes to conduct this study to further explore how it can be improved.

1.1 Research Goals and Project Scope

On a high level, the goal of this thesis is to research how the distributed systems that host Neo4j's graph databases can become more reliable. As distributed systems is a wide topic there are undeniably multiple parts of Neo4j's existing architecture that could be optimized to improve the overall reliability. To reduce the scope of this study, we will solely focus on failure detection algorithms as they have the potential to contribute to a more reliable dis-

tributed system in multiple ways. For example, by detecting failures faster, by determining the state of the machines more accurately, or by reducing the network load. Ultimately, the goal of this project is to provide Neo4j with a recommendation of how they can improve the performance of their current failure detector by either optimizing its current implementation or by replacing it with a new algorithm. In order to accomplish this goal, the existing failure detection algorithm in the current version of Lighthouse will be used as a baseline to answer the following research questions:

1. What are the strengths and weaknesses of Baseline?
2. Which alternative algorithms exist and how do they compare to Baseline?
3. How can Neo4j improve their usage of failure detection algorithms?

1.2 Related Work

The decisions that steered the development of the algorithm improvements in this study were mainly based on insights derived from previous research. Therefore, this chapter will provide an overview of what has already been studied in the field of failure detection algorithms.

Classification of Failure Detectors

A classification of failure detectors was presented by Tushar Deepak Chandra and Sam Toueg in 1996 [5]. The authors proposed a new unique solution to the consensus problem that has paved the way in the field of failure detectors since then. More specifically, their so-called "unreliable failure detector" was allowed to make mistakes while the other failure detectors at the time forcefully stopped members that were suspected, but not proven, to be faulty. Furthermore, Chandra and Toueg introduced two key concepts to classify failure detectors: completeness and accuracy. With different strengths of completeness and precision of accuracy, Chandra and Toueg made it easier to compare failure detectors by classifying them into eight different classes.

The SWIM Algorithm

Das et al. [6] introduced three extensions to improve an algorithm that they refer to as basic SWIM. The new extensions were evaluated in three configurations, where the first configuration had one extension, the second one had two extensions, and so forth. The authors evaluated the extensions by using multiple PCs where there was one node per PC and they were connected via Ethernet. The main focus of the study was scalability as they evaluated first detection speed and message load (number of received and sent messages) while changing the cluster size. In conclusion, the study found that the false positive rate was decreased with one of the extensions and the first detection speed and message load per node are independent of the cluster size.

The Lifeguard Improvements

Dadgar et al. [7] proposed three improvements to the updated SWIM algorithm presented in the previous section, called Lifeguard. A node's time-outs in SWIM are updated to change during the execution depending on events that occur to the node, e.g. an unsuccessful ping or the number of received messages. Therefore, in Lifeguard, the nodes are adapting their time-outs based on their health. The last improvement made by them was a new prioritisation of which messages to propagate first. The improvements are compared against the SWIM algorithm in two ways, one by one and as a group. The result of the study showed that the false positive rate is decreased when the extensions are applied individually and the biggest decrease is when all three improvements were activated at the same time.

The Phi Algorithm

The failure detector φ (phi) created by Hayashibara et al. [8] is a heartbeat-based algorithm, i.e. a node sends information regarding itself that it is healthy to the other nodes on decided occasions. The new detector is compared against two other ones, Chen-FD and Bertier-FD. The parameters for the algorithms compared in the study are optimised by varying them and plotting the combinations on a two-dimensional graph based on the achieved detection time and mistake rate, see Figure 1.1. For each algorithm, the optimal combination of parameters is the one whose combination on the graph is the closest to the origin, as both a low detection time and a low mistake rate are two desired characteristics. The study's result shows that the new φ failure detector achieves a comparable result to Chen-FD and Bertier-FD.

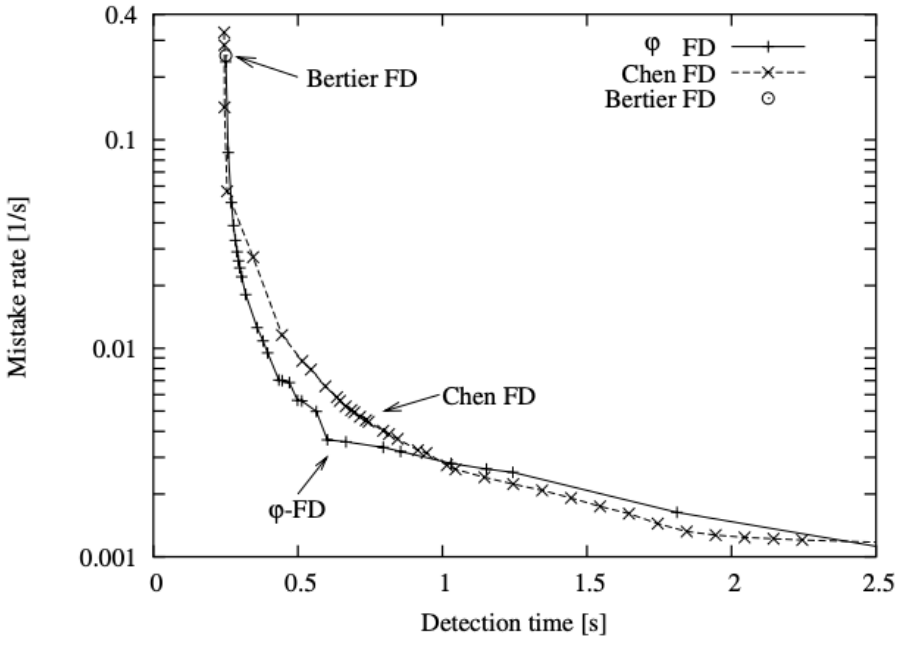


Figure 1.1: The original graph from [8] where each combination of parameters is plotted for the three investigated failure detectors.

1.3 Scientific Contribution

The scientific contribution of this thesis is mainly to provide a holistic overview of the process of choosing, implementing, simulating, and evaluating failure detection algorithms. Hence, it is not to go into the finer details of the algorithms per se as this has already been done in previous work. Instead, the focus of our thesis is to understand how the algorithms' key characteristics influence their performance. Furthermore, when starting this project and studying related work, we experienced a lack of transparency into how to actually structure and develop both the failure detection algorithms and the simulation engine in order to make quantitative evaluations and data-driven decisions. Hence, we were motivated to fill this gap by contributing with the information that we wished that we had ourselves when starting this journey. Additionally, the contribution to Neo4j has been to apply previous research about failure detectors in their codebase and evaluate them in Neo4j's particular domain.

1.4 Disposition

The aim of the following chapters is to provide a holistic overview of how to implement and evaluate failure detectors. Hence, the disposition of the thesis is as follows:

- **Introduction:** We start off by introducing the problem description from Neo4j and which research questions we aim to answer. Furthermore, we introduce related work, our scientific contribution, and how we have divided our work.
- **Theory:** This chapter aims to provide all of the necessary theory to understand the upcoming method, implementation, results, discussion, and conclusion.
- **Method:** This chapter will describe our approach to solving the given problem so that it can be replicated by someone else. More specifically, the three phases of the project will be described in greater detail: research, implementation, and evaluation.
- **Implementation:** As a major part of this thesis project consisted of practical work, this chapter will in greater detail describe what was implemented and why. More specifically, the chapter will be divided into two parts. Firstly, the overall architecture of the implemented algorithms will be described. Secondly, how the Testbed, i.e. the experimental setup, was adjusted to be able to evaluate the algorithms.
- **Results:** This chapter will provide the quantitative results that were achieved from running the experiments on the implemented algorithms in the Testbed.
- **Discussion:** After the results have been presented, this chapter will interpret and discuss them. More specifically, the focus of the discussion is to understand the impact that different implementations had on the end result. Furthermore, the limitations of the study and threats to its validity will be discussed.
- **Conclusion:** Lastly, the thesis will be concluded by answering the original research questions and presenting potential ideas for future work.

1.5 Contribution Statement

The work of the thesis has been divided equally between the two authors and it has been a close collaboration. During the literature review, both authors studied the key papers needed for conducting the research. Further on, the authors also read multiple different papers to expand the research. The majority of the implementation was performed with pair programming but the authors occasionally implemented different features alone. Throughout the writing process, the authors were assigned responsibility for different areas but it has been an iterative collaboration process. Therefore, several sections have more or less been written together. More details about individual contributions are available in Table 1.1.

Table 1.1: Estimated individual contribution.

Phase	Task	Hannes	Johan
Implementation	Algorithms	55%	45%
	Testbed	35%	65%
Report	Abstract	40%	60%
	Acknowledgements	80%	20%
	Introduction	50%	50%
	Theory	40%	60%
	Method	50%	50%
	Implementation	60%	40%
	Results	30%	70%
	Discussion	50%	50%
	Conclusion	50%	50%
	Popular Science Summary	50%	50%

Chapter 2
Theory

In this chapter, an overview of the theoretical concepts that are important to understand the use-case, importance, and evaluation of failure detection algorithms are presented.

2.1 Graph Databases

As mentioned in the first chapter, Neo4j has developed a graph database that is used by a lot of companies. In order to understand the realm of databases, a brief overview of different approaches is presented in this section.

There are many different ways to store data, e.g documents in JSON, graph databases, and relational databases [9]. Relational databases leverage several tables to store data and utilise keys (primary and foreign keys) to create relationships across tables, see Figure 2.1 [9].

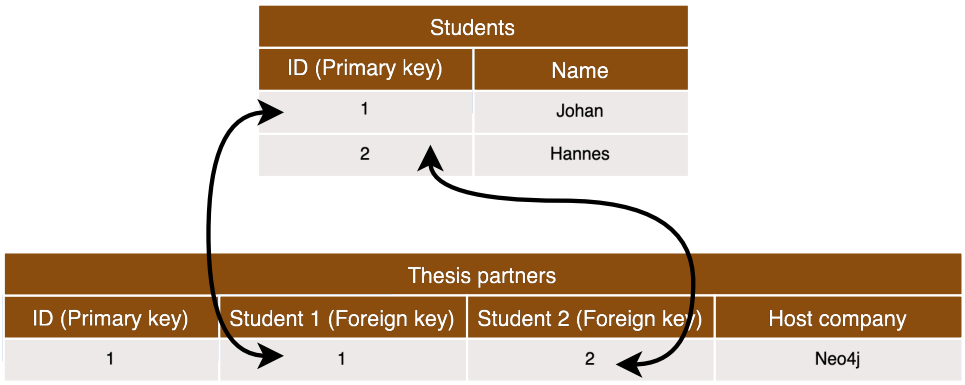


Figure 2.1: Illustration of a relationship in a relational database.

A graph database consists of nodes and relationships, see Figure 2.2 [10]. Labels can be assigned to a node and a node may also include properties [10]. A relationship between two nodes is formed with an edge with optional properties [10]. The strength of a graph database is that the relationships do not have to be calculated, since they are saved [10]. Unlike, in a relation database where the relations have to be calculated using a costly operation [10].

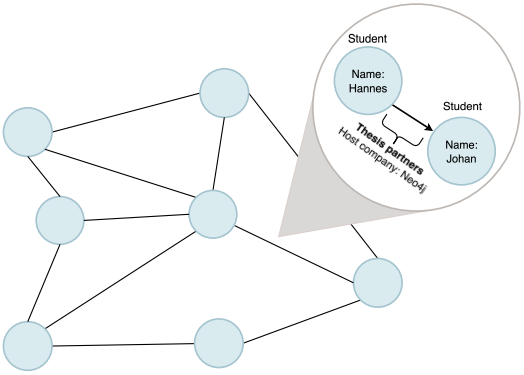


Figure 2.2: A relationship in a graph database inspired by [10].

2.2 Distributed Systems

To develop efficient failure detection algorithms it is important to have a basic understanding of the context in which they operate in, i.e. distributed systems. Van Steen and Tanenbaum define a distributed system as a "collection of autonomous computing elements that appears to it's users as a single coherent system" (p.2) [11]. As the systems are often geographically distributed, they communicate with each other through network-based messages [12]. Furthermore, Neo4j leverages distributed systems to host their graph databases, see Figure 2.3.

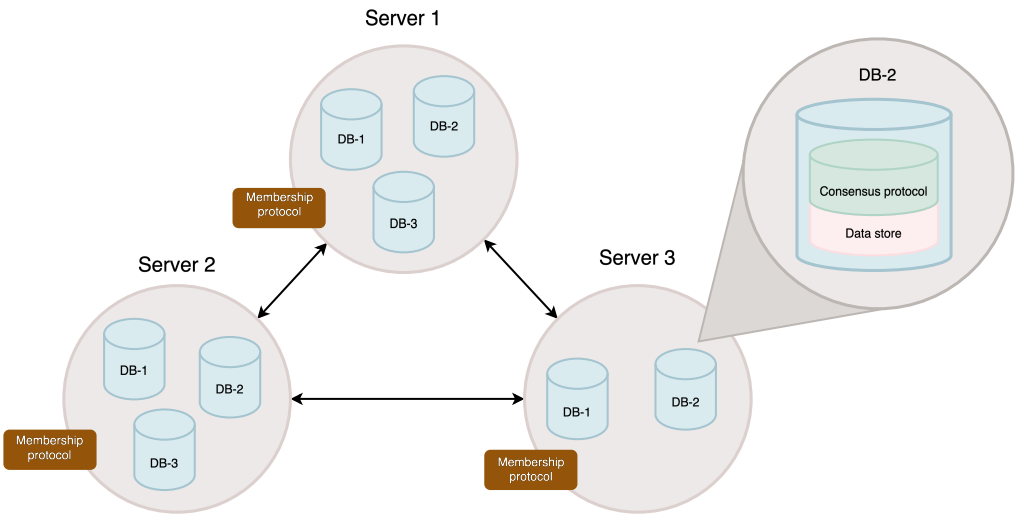


Figure 2.3: Simplified architectural overview of how Neo4j leverages distributed systems to host their graph databases.

The benefits of a distributed system are many, e.g. scalability potential, decreased costs due to shared resources, and increased reliability as it is less dependent on the failure of a single machine [2]. However, as the number of machines increases in a distributed system, so does its complexity. Some common challenges of distributed systems are unreliable networks and collaboration as every system has its own view of the system as a whole [2, 3]. The remainder of this chapter will introduce two core components of a distributed system, the consensus protocol and the membership protocol, to give context to why, where, and when failure detectors are needed.

Consensus Protocol

A so-called consensus algorithm is used to allow the members in a distributed system to agree on the state of the system as a whole [13]. While the focus of this study is not consensus algorithms per se, it is important to understand the consensus problem as both the membership protocol and the failure detector are underlyingly used to solve it. There exist multiple types of consensus algorithms that take different approaches to solving this issue and Neo4j uses the Raft protocol. In simple terms, the Raft protocol reaches consensus through voting, where each system's vote is based on their view of the system [13], see example in Figure 2.4.

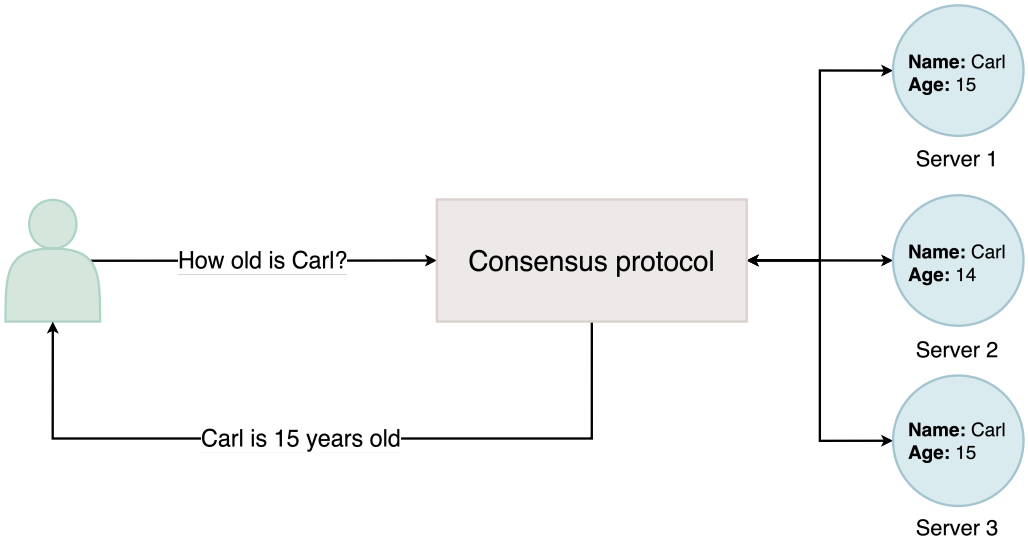


Figure 2.4: Simplified example of the consensus protocol in action when a Neo4j user gets the age from a person named Carl.

Membership Protocol

In order for the consensus algorithm to reach consensus, it must know which members exist and which ones that are working properly [14]. The membership protocol provides the consensus algorithm with this information as it tracks and updates the current states of the various members, e.g. when they leave, join, or crash [15]. As a result, the user experience of products that are built on top of distributed systems arguably "depends critically on the reliability and scalability of the membership maintenance protocol" (p.1) [6].

There exist a myriad of variations of membership protocols which over time has led to multiple ways to structure it [6]. However, on an abstract level, there are three major common components of a membership protocol that one should be aware of: a membership list, a failure detector, and a dissemination component [6]. Firstly, each machine stores its own view of its own, and the other machines, current status in a local membership list [6]. Secondly, this membership is updated based on which failures that was detected [6]. Lastly, the information stored in the membership list is communicated, i.e. disseminated, to the other machines via the dissemination component [6].

The focus of this study is membership protocols based on gossip-based dissemination. The unique characteristic of such a protocol is that the members are spreading the information together [16]. More specifically, a member spreads information by either sending a new message or forwarding a previously received one, to a random number of neighbours [16].

2.3 Failure Detection

This section will discuss common failures in distributed systems and how a failure detector can detect them. Furthermore, key properties and evaluation metrics will be introduced.

2.3.1 Failures

By design, software in a simple single system is either functioning perfectly or not at all because it is complicated for a developer to handle nondeterministic behaviour [14]. In distributed systems, however, it is inevitable that a failure will occur in some part of the system when communicating over the Internet as messages may be lost or delayed [14]. This makes failures in distributed systems more challenging as it is difficult to identify what, where, and why something failed [14]. Therefore, it is crucial to implement mechanisms that can identify and handle failures so that the system does not crash when a failure inevitably occurs [14].

Before introducing the various failure detection algorithms, it is important to understand which types of failures that they should be able to detect. To limit the scope of this project, Neo4j has identified and categorized three common types of failures in their systems that they wish to investigate: crash, link, and delay failures.

- **Crash Failure:** when a fully functional node abruptly stops working forever [5].
- **Link Failure:** when messages are lost because the link between nodes failed [17].
- **Delay Failure:** when timing issues occur due to delayed network requests [14].

2.3.2 Key Properties and Classes

In 1996, Chandra and Toueg introduced two key properties and eight classes that are commonly used to objectively compare and categorize different failure detectors, see Figure 2.5 [5]. In simple terms, completeness is a measurement of whether or not each crash failure is eventually detected by all nodes or just some [5]. Meanwhile, the accuracy property describes how trustworthy the failure detector is [5]. For example, a failure detector that fulfills the strong accuracy property never suspects a node before it actually failed [5].

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	Perfect P	Strong Y	Eventually Perfect $\diamond P$	Eventually Strong $\diamond Y$
Weak	L	Weak W	$\diamond L$	Eventually Weak $\diamond W$

Figure 2.5: Chandra and Toueg's eight classes of failure detectors [5].

2.3.3 Evaluation Criteria

To iteratively improve a failure detector, and compare alternative implementations, it is important to be able to identify its strengths and weaknesses through measurable metrics. For example, there are inevitably design choices that can be made as a failure detector could be considered "conservative", i.e. extremely accurate, or "aggressive", i.e. extremely fast [8, 18]. While there exist multiple different metrics to evaluate failure detectors, the most common ones could be grouped into three categories: speed, accuracy, and network load [6, 7]. The following six evaluation metrics were chosen to evaluate the failure detectors in this study:

- **First Detection:** The time it takes for the first node to detect that another node has crashed [6, 7].
- **Full Dissemination:** The time it takes from a crash failure has occurred until all nodes have received the information [7].
- **Undetected Failure Rate:** The percentage of all crash failures that were undetected by all nodes that are working properly.
- **False Positive Rate:** The percentage of all claimed crashes where the node was actually working properly [6].
- **Total Messages Sent:** The accumulated number of messages that the failure detector transmitted over the network [7].
- **Message Load:** The average number of bytes that the failure detector transmits over the network per second [6].

2.3.4 Algorithms

This section provides an introduction to how the investigated failure detection algorithms in this study work internally. Before diving into the various algorithms, let's introduce which algorithms popular vendors use. The authors of Lifeguard [7] implemented their own version of the SWIM algorithm called Memberlist and it is used by vendors like Serf, Nomad, and Consul [7]. Akka is also a vendor that provides failure detection capabilities with their product Akka Cluster which uses the φ failure detector for detection and a gossip protocol to disseminate information [19]. Therefore, while different vendors may have made minor modifications to the original SWIM algorithm, it is arguably a popular choice by some of the most established vendors within the industry.

2.3.5 SWIM

Since the first Basic SWIM algorithm was introduced in 2001 by Gupta et al. [20] it has been extended by Das et al. [6] with three extensions. The focus of this study is the SWIM algorithm with the three extensions. From now on when referring to SWIM, the basic SWIM algorithm together with the three extensions is meant and the rest of this subsection will explain the algorithm in greater detail.

The algorithm can be divided into two separate parts: the failure detection module which pings peers to see if they are alive, and the dissemination module which spreads updates from the failure detection module [6]. To understand how the failure detection module pings its peers, it is important to comprehend the states that the nodes can transition between [6]. More specifically, they are alive, suspected, and unhealthy, and a node can transition between these states through *Suspect*, *Alive*, and *Confirm* messages [6].

One of the key characteristics of the SWIM algorithm is that if it does not receive an answer from the initial ping, it asks its peers for help [6]. More specifically, this is done by sending a new type of request, a *ping-req*, to k random nodes [6]. The k random nodes themselves try to contact the concerned node and send the ack back to the original node in the event of a reply [6].

SWIM needs three parameters to operate, the protocol period T, the k number of peers to ask for help, and λ that limits how many times the dissemination module sends a message [6]. Furthermore, the authors also emphasise "(...) that the protocol period T' has to be at least three times the round-trip estimate" (p. 4) [6].

Now that the states and pinging mechanisms have been introduced, the workflow of the failure detection module can be understood in its entirety, see Figure 2.6.

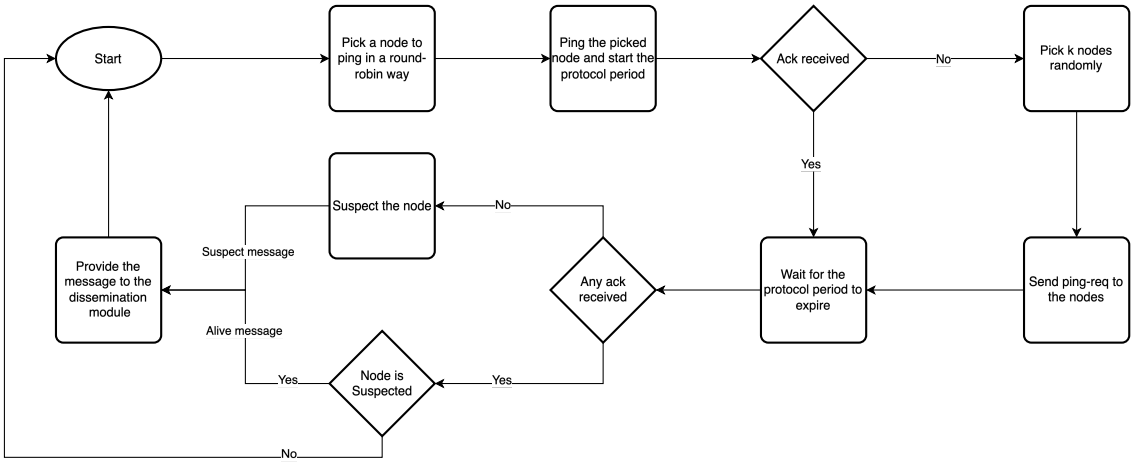


Figure 2.6: Flowchart of the failure detection module's workflow as interpreted by [6].

A key characteristic of the dissemination module is that it uses piggybacking which means that updates from the failure detection module are added on top of the existing messages [6].

Once a node becomes *Suspected* in another node's membership list, a timer is started [6]. A *Confirm* message is issued, which transitions the node from suspected to unhealthy, if the time expires [6]. On the contrary, if the timer has not yet expired and a node that is alive receives an accusation that it has failed, it will issue an *Alive* message [6].

As a node can repeatedly transition between states, the most up-to-date state is derived based on the priority of the message and the incarnation number of the node, see Formula 2.1 [6]. More specifically, an incarnation number keeps track of the most up-to-date message regarding an individual node [6].

$$\begin{gathered} \text{Alive}_j\ ,\ \text{Suspect}_j < \text{Confirm}_i \quad \forall i, j \\ \text{Alive}_j\ ,\ \text{Suspect}_j < \text{Alive}_i \quad \forall j < i \\ \text{Alive}_k\ ,\ \text{Suspect}_j < \text{Suspect}_i \quad \forall j < i\ ,\ \forall k \leq i \end{gathered} \tag{2.1}$$

Illustration of the messages' priority from [6] with subscripted incarnation number.

2.3.6 Lighthouse

This section explains Neo4j's current failure detection algorithm which is inspired by the SWIM algorithm. This section will introduce the key similarities and differences to understand the Lighthouse algorithm. Furthermore, the text and figures in this subsection are based on internal documents and the source code.

The Lighthouse algorithm comprises of the same two modules as the SWIM algorithm. The purpose of the pinging algorithm is to check whether a node is alive and disseminate the information. Unlike SWIM, Lighthouse does not ask its peers for help. Instead, the pinging algorithm works as follows:

1. Select a node from the membership list in a round-robin way.
2. Send a ping to the selected node and attach the membership list as a payload. [1]
3. Wait for the protocol period, T.
   (a) Ack received before T, merge the payload's membership list with the local list.
   (b) Otherwise, transition the selected node to *Suspected*.
4. Return to step 1.

Lighthouse introduces additional states: *Unreachable* and *Left*, see Figure 2.7 for the full state transition graph. The *Removed* state is the same as unhealthy in SWIM. *Left* is the state in which a node transitions itself to when it wants to leave the cluster of its own will, i.e. gracefully. More specifically, *Unreachable* is a pit-stop between *Suspected* and *Removed* to give the node a second chance to refute the suspicion at a later time. An important note, from now on when referring to a failed, crashed, unhealthy, etc node in Lighthouse, the Unreachable state is meant.

[1] Note when a node receives a ping it merges the payload of the membership list with its local one.

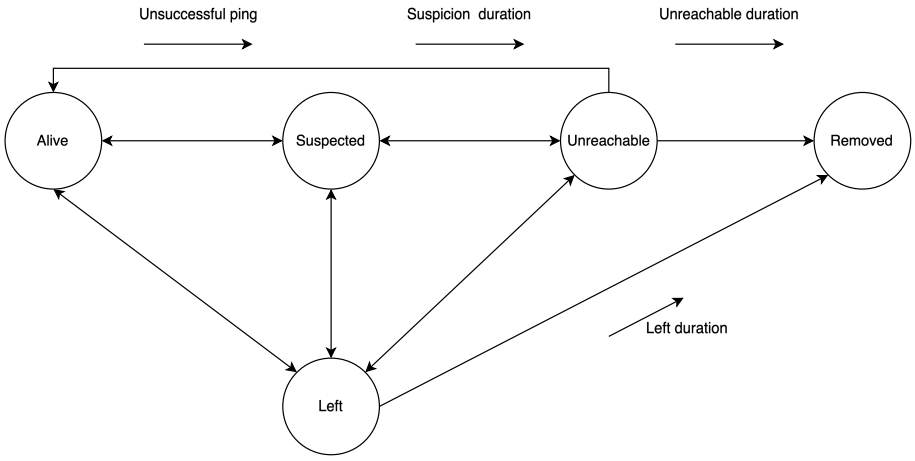


Figure 2.7: Illustration of the state transitions in Lighthouse.

Similar to SWIM, the states in Lighthouse also have a priority which determines if a node's state in the local membership list should be overwritten or not. Firstly, the states are based on the incarnation number. A higher incarnation, the higher priority of the state. In the case of equal incarnation numbers, the state determines the priority, see Formula 2.2.

$$\text{Alive} < \text{Suspected} < \text{Unreachable} < \text{Left} < \text{Removed} \tag{2.2}$$

Similar to SWIM, all nodes have timers for state transitions that either can be expired or invoked. However, additional timers have been added for each of Lighthouse's new states, see Figure 2.7 and Table 2.1.

Parameter	Time
Protocol period	1s
Suspicion duration	30s
Unreachable duration	24h
Left duration	24h

Table 2.1: Parameters that were initially used in Lighthouse.

Similar to SWIM, Lighthouse utilises piggybacking to spread updates around the cluster. However, an important difference is that Lighthouse always sends its full membership list while SWIM only sends the updated states.

Chapter 3
Method

The project was divided into three phases: research, implementation, and evaluation. The aim of this chapter is to describe each phase in greater detail to provide a clear overview of what has been done and why.

3.1 Phase 1: Research

The first phase of the thesis was spent on researching the area and further specifying the scope of the project. The related work that was presented in Section 1.2 was identified and closely studied to survey new potential algorithms. Furthermore, internal documentation and source code of the existing Lighthouse implementation were studied to better identify strengths and weaknesses. After discussing the literature and limitations of the existing solution with Neo4j, it was decided that the focus of the study should be on developing the SWIM algorithm presented in [6] as it showed great synergy potential with Neo4j's existing codebase. Furthermore, based on the parameter tuning approach presented in [8], it was decided that we should try to improve Baseline by finding the best parameters for both Lighthouse and SWIM.

3.2 Phase 2: Implementation

In the second phase of the thesis, the new SWIM algorithm was implemented and the existing test environment, i.e. the Testbed, was modified. The SWIM algorithm was implemented in Java (Temurin-17.0.10+7) inside of Neo4j's existing codebase. The functionality of SWIM was verified by developing unit tests and by manually simulating different scenarios and inspecting logs to verify the intended behaviour. An evaluation engine was developed in Python 3.9 on top of Neo4j's existing Testbed to be able to handle identified issues with the test environment that was identified in Phase 1.

3.3 Phase 3: Evaluation

The evaluation phase consisted of five parts. Firstly, an evaluation criteria was defined. Secondly, an experimental setup was defined to allow consistent simulations. Thirdly, a Baseline configuration was established. Fourth, two additional configurations were established by optimising the parameters for SWIM and Lighthouse, named Optimal SWIM and Optimal Lighthouse. Lastly, the three configurations were compared.

3.3.1 Evaluation Criteria

An evaluation criteria was established by identifying multiple relevant metrics in related work and selecting the most appropriate ones for Neo4j's use-case, see Table 3.1.

Table 3.1: Evaluation criteria.

Evaluation Category	Evaluation Metric
Speed	Average Full Dissemination Average First Detection
Accuracy	Undetected Failure Rate False Positive Rate
Network Load	Total Messages Sent Message Load

3.3.2 Experimental Setup

All of the experiments were run on an Apple MacBook Pro M1 Max from 2021 with 64 GB of memory. During the experiments, unnecessary programs were terminated to minimise the inference with the simulator. Discussions with Neo4j guided the definition of the experimental setup to mimic the real networks that they operate in, see Table 3.2.

Table 3.2: Parameters used for the experimental setup. Note that the values within brackets are randomly generated within the interval.

Parameter	Tested Values
Number of nodes	10
Nodes to kill (%)	40
Interval between kills (ms)	[2500, 5000]
Interval between disturbance (ms)	[3000, 5000]
Interval disturbance length (ms)	[1000, 2000]
Latency (ms)	800
Jitter (ms)	[300, 500]
Probability of link failure (%)	10

3.3.3 Establishing a Baseline

A Baseline was established to be able to compare the impact of our work and make a data-driven recommendation. More specifically, Baseline was established by running three simulations for the original Lighthouse failure detector and calculating the average score for each metric. The parameters used for Baseline were taken from the original source code and are displayed in Table 3.3.

Table 3.3: Baseline configuration based on the original Lighthouse failure detection algorithm in the original source code.

Parameter	Tested Values
Protocol period (ms)	1000
Suspicion duration (ms)	30000

3.3.4 Parameter Tuning

The best parameters for Lighthouse and SWIM were derived by simulating and evaluating different combinations of parameters, see Table 3.4. The tested parameters were decided together with Neo4j based on the previously defined experimental setup, see Table 3.2.

Table 3.4: Overview of the parameters that were evaluated during the parameter tuning process. In total, there were 33 tested combinations, 15 for Lighthouse and 18 for SWIM.

Configuration	Parameter	Label	Tested Values
Lighthouse	Protocol period (ms)	T_l	{1000, 1500, 2000}
	Suspicion duration (ms)	S_l	{10000, 15000, 20000, 30000, 40000}
SWIM	Round-trip time (ms)	RTT	{1000, 1500, 2000}
	Nodes to probe	k	{2, 4}
	Lambda	λ	{5, 10, 15}

The best parameters for each algorithm were decided based on which combination that generated the lowest parameter optimisation score based on the average first detection speed and false positive rate from three simulations. More specifically, the parameter optimisation score was a slight modification of the formula used in [8] as we chose to introduce importance weights, see Formula 3.1.

$$\text{score}(x, y, w_x, w_y) = \sqrt{w_x \cdot x^2 + w_y \cdot y^2} \tag{3.1}$$

After multiple discussions with Neo4j, the two most important metrics for their use case were the average first detection speed and the rate of false positives. Hence, x is the first

detection speed and y is the false positive rate. Furthermore, the average first detection speed was considered to be three times more important than the rate of false positives. Therefore, we want to penalise large values of false positives by selecting $w_y = 3$ and $w_x = 1$. Note that a lower score indicates a better result as a low rate of false positives and an average fast detection time is desired. In order for both metrics to have the same order of magnitude, the false positive rate was changed to decimal form and the detection speeds were divided by the maximum detection speed.

3.3.5 Evaluating Optimal Configurations

After finding the optimal configurations of Lighthouse and SWIM, they were evaluated against Baseline in three ways. Firstly, all metrics from the evaluation criteria presented in Table 3.1 were compared. Secondly, the limits of the algorithms were evaluated by simulating them in a network more prone to link failures, from an occurrence rate of 10% to 50%. Thirdly, the scalability of the various algorithms was evaluated by varying the number of nodes in the cluster from 10 to 75. During the scalability investigations, VisualVM was also used to analyse the number of active threads. For the three types of evaluations mentioned in this section, three simulations were run and average metrics were calculated to mitigate outliers.

Chapter 4
Implementation

The implementation of Lighthouse by Neo4j is explained in this chapter. Further on, the chapter provides an overview of how SWIM was implemented and how the existing Testbed was improved to be able to simulate failures and provide essential evaluation metrics.

4.1 Lighthouse

The implemented Lighthouse failure detector by Neo4j consists of three main parts that each member has: incoming traffic, outgoing traffic, and protocol clients, see Figure 4.1.

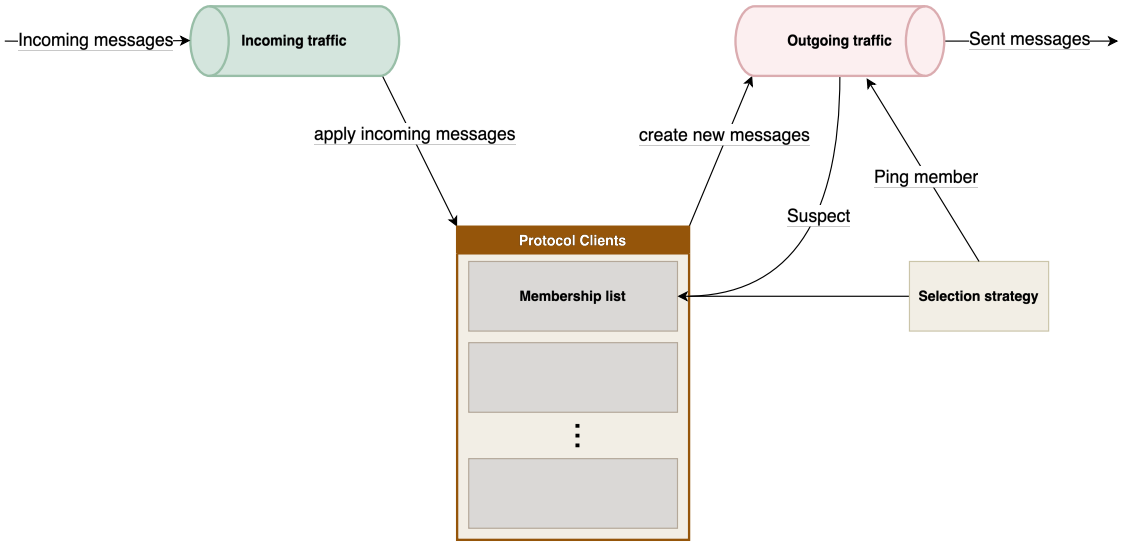


Figure 4.1: Lighthouse node structure.

The incoming traffic and outgoing traffic components run in two separate threads and are connected by a list of protocol clients which all implement a common interface. The two

most important functions of the protocol clients are *apply* and *newMessage*. Apply is used by the incoming and outgoing traffic components to apply the payload elements to a protocol client. Note that the implementation of the function can be different for each protocol client. The purpose of *newMessage* is to ask a protocol client if it has a payload element to send.

Lighthouse has a membership list implemented as a protocol client that keeps track of the node's local view of other nodes' states and schedules state transitions. The protocol client membership list also implements other interfaces to support more functionality. The function *newMessage* for this protocol client returns a copy of the local full-state view of the node. The *apply* function on the other hand requires a full state view and merges it with its own local view based on the priority of the states and incarnation number.

The incoming traffic's responsibility is to receive messages, decode them, apply the correct function depending on the message type, and respond with an acknowledgement (ack). The component of incoming traffic iterates through the payload elements of the message, see Figure 4.2 for message structure, and calls the *apply* method on the correct protocol client.

The role of the outgoing traffic component is to ping other nodes in the cluster. The outgoing traffic has reference to the protocol clients, a selection strategy, and a listener for suspected nodes. The previously mentioned protocol client acts as the listener for suspected nodes. The outgoing traffic starts by asking the selection strategy for which node to ping. The selection strategy keeps track of the recently picked nodes and therefore has a connection to the membership list to be able to know what nodes exist. Before sending the ping, the outgoing traffic iterates through the protocol clients and asks if they have something they want to send (by calling *newMessage*) and encodes it. After the ping is sent, it waits for an ack and if it receives the ack, the payload is processed by the correct protocol client. If not, it forwards the information to the suspect listener, the membership list, that the node is *Suspected*.

When a Lighthouse node starts, the membership list is initialised with the addresses of the nodes in the initial cluster and the outgoing and incoming traffic threads are started. However, before that, a thread tries to send join requests to the other nodes and continues to do so until it has information regarding all nodes' states from the initial cluster. During a join request, the sending node sends its entire membership list and the receiving node responds with its entire membership list.

The payload sent in Lighthouse follows a specific structure and consists of three fields, see Figure 4.2. Payload elements are sent in the payload by Lighthouse and the content of the payload elements is shown in Figure 4.3.

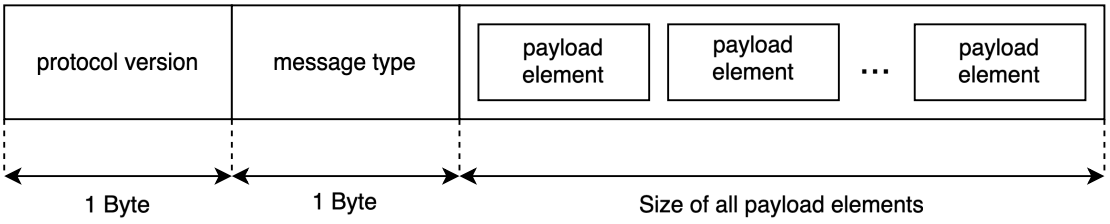


Figure 4.2: Structure of payload sent by Lighthouse.

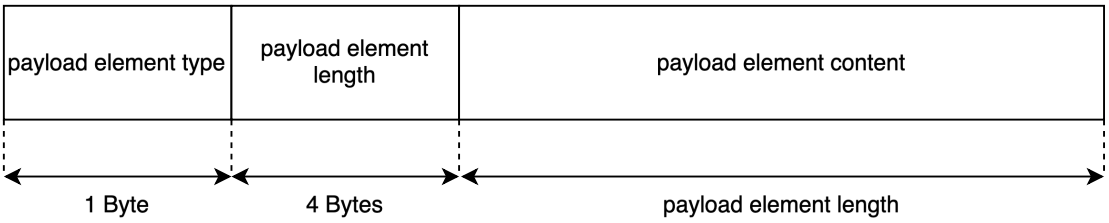


Figure 4.3: Structure of payload element.

4.2 SWIM

In Section 2.3.6 it is explained that Lighthouse is to a large extent based on SWIM. However, they differ in the way pinging is performed, the states, and what information is sent through the cluster. During this thesis project, the SWIM algorithm was implemented in the existing codebase where Lighthouse already existed. Hence, the architecture and code of the existing Lighthouse were re-used as a foundation for the new SWIM algorithm. In the following sections, it is explained what changes were made in order to implement SWIM. The structure of the coming subsections corresponds to the parts of a Lighthouse node.

4.2.1 Outgoing Traffic

Lighthouse had already implemented the functionality of selecting members to ping in round-robin. A new function was added to the selection strategy class to be able to pick an arbitrary number of random members. A seed was added to the selection strategy to ensure that the same nodes are pinged in different simulations.

The outgoing traffic component was updated to implement the feature of probing k neighbours if the first ping was unsuccessful. A new parameter for the SWIM algorithm was introduced called round-trip time, $\boldsymbol{RTT}$, and it is defined as the maximum time to wait for an ack after sending the first ping. If $\boldsymbol{RTT}$ expires then the process of probing k neighbours starts. We defined the protocol period as $3.1 \cdot \boldsymbol{RTT}$ based on the recommendation from the SWIM paper[6], that it should be more than three times bigger. The probing of neighbours was implemented by creating a new thread for each *ping-req* sent and the maximum time of waiting for an indirect ack is the remaining time of the protocol period.

After each thread has terminated, it returns a boolean value indicating if it received an ack, and the threads were examined if at least one of all the threads received an ack. In the case of no threads receiving an ack, then the node is determined to be *Suspected* and the suspect function of the suspect listener is executed. However, if the first ping or the indirect pings are successful, the node is alive and a new function called alive on the suspected listener is executed.

The *ping-req* messages required a new message type and modification of the structure of the payload, see the new structure in Figure 4.4. To solve the problem of knowing what node to ping a new optional field was added to the payload containing the address of what node to ping.

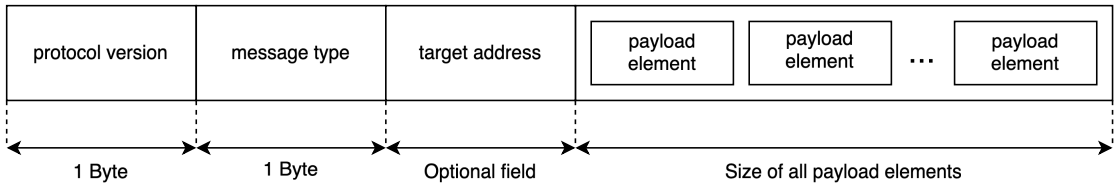


Figure 4.4: Structure of payload element in SWIM.

4.2.2 Incoming Traffic

The incoming traffic component of SWIM is similar to Lighthouse's except for the fact that SWIM has to handle a new *ping-req* message. A new case was added to match the new message type that applies the payload to a new function. The new function consists of several steps. The first step is to get the target address of the node to ping, with the new field of the payload element. The function builds a ping message by asking the protocol clients if they have anything to send, pings the target address, and waits ***RTT*** for the ack. If an ack message is received before the timeout expires, the payload of the ack is applied to the correct protocol client, and an ack response is formed for the node that sent the *ping-req* message in the first place.

4.2.3 Membership List

The original Lighthouse membership list was modified for the SWIM algorithm. Firstly, the states were updated by removing the *Unreachable* state, but the *Left* state was kept despite the fact it is not part of the SWIM algorithm, see Figure 4.5. The reason behind keeping the *Left* state was to support the graceful leave. In the implementation, the full membership list is not passed around, but rather selected messages. The states correspond to the different types of messages sent in the SWIM algorithm, except *Left*. *Alive* corresponds to an *Alive* message, *Suspected* to a *Suspect* message, and *Removed* to a *Confirm* message. A new compare function between the states was implemented to support the priority of the messages from SWIM.

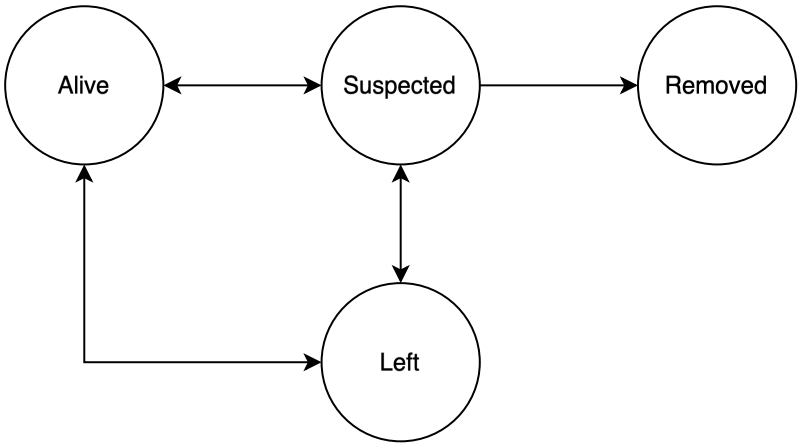


Figure 4.5: State transition graph in SWIM implementation.

The *Alive* message that is created when successfully pinging a suspected node was implemented by introducing a new function alive in the member listener interface. Every time the

outgoing traffic successfully pings a node it executes the alive function on the membership list. The membership list creates this message only if the pinged node is suspected in its local list. The new message was implemented by introducing an additional field in the class regarding the state of a node, to indicate this type of message. When this message is received by a node's membership list it overrides every state except a removed state.

The SWIM algorithm propagates selected messages around the cluster instead of sending the whole local membership list around the cluster as in Lighthouse. It is implemented using a wrapper class around a priority queue with a custom comparator regarding the number of times the message has been sent, as instructed by [6]. The authors of SWIM used a buffer for their implementation [6]. When either the incoming or outgoing traffic calls the function *newMessage* that function in turn calls another function that iterates through the queue, increments the messages that are sent, and removes messages that have been sent the maximum number of times.

The authors of the SWIM algorithm [6] explained that the number of times a message is sent is $\lambda \log n$. However, during the evaluation, the following was used by the authors $3\lceil \log(n+1) \rceil$, and they defined the suspicion duration in seconds as that value [6]. Based on the previous, the suspicion duration in seconds and the number of times a message is sent were defined by Formula 4.1.

$$\lambda \cdot \lceil \log_{10}(n+1) \rceil \tag{4.1}$$

With the new membership list and the priority queue, when the incoming and outgoing traffic calls *apply* to the membership list, the state messages are merged with the local membership list if the state from the message has a higher priority. If the latter applies, the messages are added to the local priority queue and the messages will be forwarded in the future.

4.2.4 Configuration and Building

During the implementation of SWIM, we wanted to structure the implementation in the same way as Lighthouse and try to reuse as many parts as possible for easier maintenance and less duplicated code. To achieve it, we reused the same interfaces that were used for configurations of parameters for Lighthouse and added optional fields for the SWIM-specific parameters. Further on, selected interfaces were updated to be generic to support the different types of the two algorithms. In the same spirit, the builder class for Lighthouse was updated so that it could build both algorithms. This was primarily done for the reason of the Testbed. Instead of having two different types for the algorithms, one common type was used and it limited the changes that had to be made to accommodate the new SWIM algorithm to run in the existing Testbed.

4.2.5 Verification of SWIM

The verification of SWIM was completed in two parts. Firstly, the membership list for SWIM was tested with modifications of Lighthouse's unit tests along with new unit tests specific to SWIM. Secondly, the communication between the nodes, the pinging, was verified manually. The source code was changed during the verification so that all ack messages from a selected

node in the first ping always resulted in probing k nodes, to force the indirect pings. The logs were examined so that the picked node was not determined to be *Suspected* or *Removed*, since it was healthy. A second run was made in which the picked node was killed via an HTTP request, and the logs were examined once again to see that the node was removed by the nodes in the cluster.

4.3 Testbed

The aim of this section is to provide an overview of how a testing environment can be structured to simulate a distributed system and evaluate failure detection algorithms. To provide a transparent overview of the work that was made, we will first describe the original Testbed and then the modifications that were made.

4.3.1 The Original Testbed

On an abstract level the original Testbed consisted of four main parts: a Configuration generator, Test cluster, Disruption generator, and a Monitor, see Figure 4.6. The purpose of the Configuration generator is to generate all of the configuration files needed by the Test cluster whose job is to start an HTTP server for each individual node. Furthermore, the Test cluster has an API that allows the nodes to be killed through HTTP requests. The nodes communicate with each other via a middleman, the Disruption generator, whose job is to simulate a real network with latency, jitter, etc [21]. Meanwhile, the monitor collects data from the test cluster and visualises it through Prometheus, a time-series database that scrapes the collected data from the test cluster and visualises them in real-time [22].

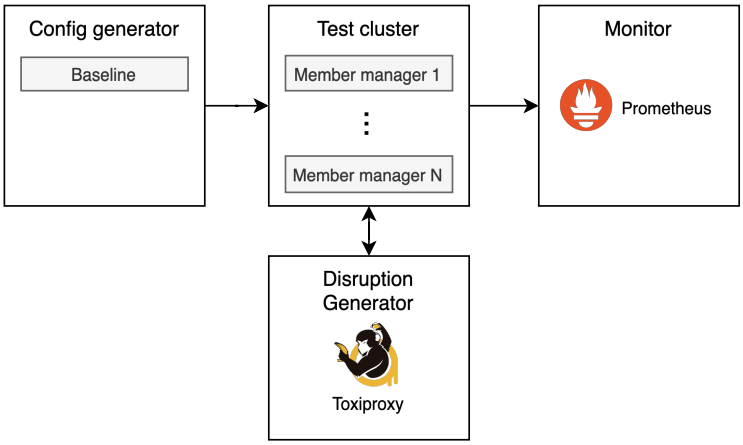


Figure 4.6: Overview of the original Testbed's architecture.

The primary goal of the original Testbed was basic debugging and verification that the system behaved as expected. E.g. by killing a node through an HTTP request and then looking inside Prometheus if the number of alive nodes was decreased by one. However, three weaknesses were identified in the existing Testbed that needed to be improved: scalability, consistency, and metrics.

Weakness 1: Scalability

Running a simple simulation in the original Testbed was cumbersome and error-prone as it required a lot of manual steps. In simple terms, the original workflow was as follows:

1. Build the project to include the latest version.
2. Start Prometheus via Docker to monitor the processes.
3. Enable and start-up Toxiproxy, the disruption service.
4. Build the configuration.
5. Start the cluster.

Performing these steps may not look too complex. However, all these five steps had to be done to run a single test. Furthermore, the original Testbed only had support for running Baseline as it was hardcoded in the source code. If you wanted to try other parameters you had to manually change it in the source code, build the code, and re-run the Testbed, i.e. re-do all five steps mentioned above. The original Testbed lacked scalability as it only allowed one, pre-defined, configuration to be evaluated. Furthermore, running simulations was an extremely time-consuming process which would make it difficult to simulate multiple configurations to find the optimal parameters for each algorithm.

Weakness 2: Consistency

In the original Testbed, it was not possible to run the exact same simulation multiple times. Therefore, it was difficult to evaluate different algorithms under equal conditions which made it impossible to draw an objective conclusion. More specifically, this was because the original Testbed required manual HTTP requests to kill nodes.

Weakness 3: Evaluation

The original Testbed was mainly used for manual troubleshooting, not evaluation. The collected metrics that were visualised in Prometheus did not provide sufficient information to calculate all of the evaluation metrics described in Section 2.3.3. More specifically, the collected data was more related to the total network load and the number of nodes in each state, not which node was in which state at what time.

4.3.2 The Modified Testbed

Several modifications were made to the original Testbed, see the architectural differences in Figure 4.6 and 4.7. This section will in greater detail describe the modifications that were done to solve the previously identified weaknesses: scalability, consistency, and evaluation.

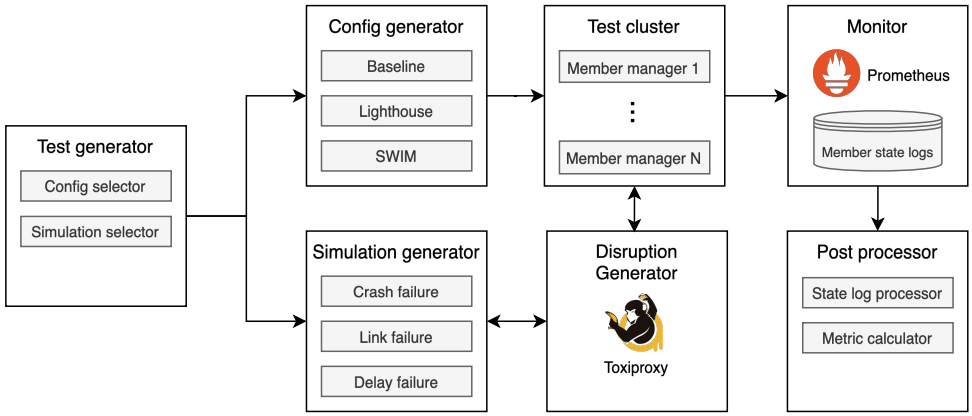


Figure 4.7: Overview of the modified Testbed's architecture.

Solution for Weakness 1: Scalability

A Test generator was built on top of the original Testbed to improve the user experience and allow systematic testing. More specifically, it automatically started up all processes by starting the program by just running the program and removing the need to re-do the five steps described in Section 4.3.1. The Test generator also allowed the user to define which algorithms and parameters they would like to run. To allow this, modifications were done to the Configuration generator and Test cluster so that parameters could be sent to the Configuration generator via the terminal and that the Test cluster could read these parameters in the configuration files and apply them to the builder of the algorithms. As a result, it was now possible to do multiple simulations in one click so that one could start a longer simulation and let it run in the background.

Solution for Weakness 2: Consistency

A Simulation generator was developed to systematically trigger failures in the network that simulated Neo4j's real networks. More specifically, it allowed the triggering of crash, link, and delay failures systematically instead of sending manual HTTP requests. The occurrence and duration of each simulated failure were defined so that they appeared consistently in all runs of the simulation, see the pre-defined experimental setup variables in Table 3.2. For all randomly generated values, a seed was used to make sure that the same random value was generated every time the simulation ran to ensure an equal testing environment.

The crash failure and the delay failure were executed in two separate threads in the simulation generator to allow multiple failures to occur simultaneously. To simulate a crash failure, a random node is killed by sending a kill request to the Test Cluster's API. After the request is sent, the threads wait for a random time and repeat until the pre-defined number of nodes that should crash have been killed. The delay failure was implemented in a similar way as HTTP requests were sent to Toxiproxy which instructed it to add a random jitter to the latency for a random node. After a random time, another HTTP request is sent to Toxiproxy to reset the jitter. Lastly, the thread waits a random time before it repeats the process for another node.

The link failure was implemented by modifying the network protocol used by the nodes in the Test Cluster to send messages to other nodes. More specifically, the function used for

sending the messages was updated to only send the packets if a random value between 0 and 1 was smaller than the probability of link failure, otherwise, no message was sent at all.

Solution for Weakness 3: Evaluation

Member state logs were added by modifying the protocol client of both algorithms so that they logged their local membership lists on every state change. Further on, logging was also made when a suspicion duration expired, i.e. when a node transitioned from *Suspected* to *Removed*. Finally, A Post-processor was implemented that consisted of two sub-components: a state log processor and a Metric calculator. The state log processor reads all of the new logs and derives them into more tangible data that the Metric calculator can use to calculate the six key evaluation metrics, see Figure 4.8.

Killed node	Actual kill time
2	2024-04-18 14:35:01
3	2024-04-18 14:35:04
...	...

(a) Actual kills table. Registers the time a kill request was sent from the Simulation generator to the Test cluster API.

Failed node	Timestamp
2	2024-04-18 14:55:53
4	2024-04-18 14:57:02
...	...

(b) Suspicion expiration table. Each node has its own table where it saves the timestamp and node that it has personally removed.

Timestamp	Node 1	Node 2	...	Node N
2024-04-18 14:34:59	Alive	Alive	...	Alive
2024-04-18 14:35:02	Alive	Suspected	...	Alive
2024-04-18 14:35:05	Alive	Removed	...	Alive
...	...	...	...	...

(c) Membership list table. Each node has its own table where it saves its local view of the other nodes. The information in this table is not necessarily detected by the node itself as it may have been gossiped from peers.

Killed node	Actual kill time	Detected kill time		
		Node 1	...	Node N
2	2024-04-18 14:35:01	2024-04-18 14:35:05	...	2024-04-18 14:35:03
3	2024-04-18 14:35:04	2024-04-18 14:35:07	...	2024-04-18 14:35:05
...	...	...	...	...

(d) Detected kills table. Data from a) and c) are compiled to see differences in actual kill time and detected kill time. E.g Node 1 detected that Node 2 was killed 4 seconds after it was actually killed.

Figure 4.8: Data structures created by the State log processor.

With the help of the data structures from the State log processor and the queried network data from Prometheus, the Metric calculator calculated the key evaluation metrics in the following way:

- **Average First Detection:** For each actual kill, i.e. for each row in Figure 4.8d, get the earliest detected kill time and subtract the actual kill time. Calculate the average first detection time by taking the sum of all first detection times and dividing it by the number of kills, i.e. number of rows.
- **Average Full Dissemination:** For each actual kill, i.e. for each row in Figure 4.8d, get the latest detected kill time and subtract the actual kill time. Calculate the average full dissemination time by taking the sum of all full dissemination times and dividing it by the number of kills, i.e. number of rows.
- **Undetected Failure Rate:** The number of undetected failures is equal to the number of rows in Figure 4.8d where no node detected the kill, i.e. where all detected kill times are empty. Hence, the undetected failure rate is the total number of undetected failures divided by the total number of kills, i.e. rows in Figure 4.8a.
- **False Positive Rate:** The number of false positives is the number of occurrences where an identified kill in Figure 4.8b was actually not reported dead in Figure 4.8a or if the timestamp from the identified kill in Figure 4.8b happened before the node was actually killed in Figure 4.8a. Lastly, the rate of false positives is the number of false positives divided by the number of times the suspicion duration expired.
- **Total Messages Sent:** The original Testbed periodically submits the number of messages sent by each node to Prometheus. Hence, the total message sent by each node is calculated by calculating the difference between the number of messages sent before the simulation has started and the number of messages sent after the simulation has finished. Lastly, the total messages sent is calculated by adding the total number of messages sent by each node.
- **Message Load:** The original Testbed periodically submits the number of bytes sent by each node to Prometheus. To derive the message load, i.e. the bytes sent per second, the total bytes sent are first calculated by deriving the difference from the number of bytes sent by each node at the start and end of the simulation. Afterward, the total bytes sent for all of the nodes are summarised and divided by the length of the simulation.

Chapter 5

Results

This chapter will present the results from evaluating SWIM against Lighthouse. The results will be divided into two parts: parameter optimisation and further investigations. In part one, the most suitable parameters for SWIM and Lighthouse will be derived. In part two, Optimal Lighthouse and Optimal SWIM will be further evaluated against Baseline by investigating how well they perform in networks more prone to link failures and in clusters of various sizes.

5.1 Parameter Optimisation

There were in total 18 parameter combinations tested for SWIM and 15 for Lighthouse, see Table 3.4 for more details. As described in the methodology in section 3.3, each parameter combination was evaluated based on their average first detection speed and false positive rate. The obtained values for each parameter combination are plotted in Figure 5.1. A noteworthy observation in Figure 5.1 is that the majority of the Lighthouse parameter combinations have a false positive rate close to zero while SWIM shows greater variation.

While Figure 5.1 visualises the average obtained values for each of the two key metrics, Table 5.1 displays the actual result from the parameter optimisation. More specifically, the identified parameters that formed the two optimal configurations, Optimal SWIM and Optimal Lighthouse, are based on the best score. A lower score is desired as it means that the algorithm is fast at detecting failures while generating a low rate of false positives, see Formula 3.1. Both Optimal Lighthouse and Optimal SWIM achieved a better score than Baseline, see Table 5.1.

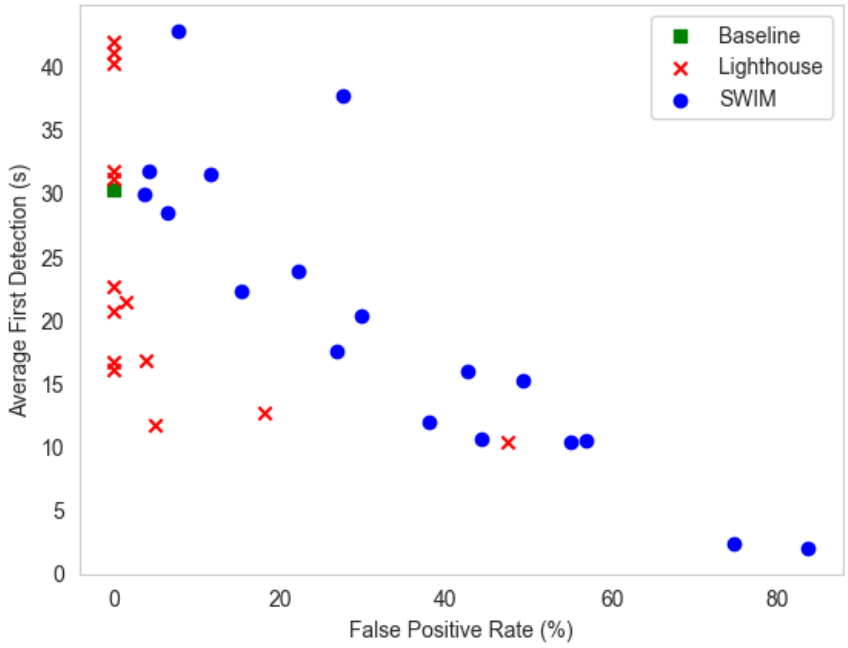


Figure 5.1: False positive rate and average first detection time for each unique parameter combination presented in Table 3.4. The scores are based on the average from running three simulations.

Table 5.1: The chosen parameters for Baseline, Optimal Lighthouse, and Optimal SWIM. The parameters for Baseline were gathered from the original source code. Meanwhile, the parameters for Optimal Lighthouse and Optimal SWIM were chosen based on which combination of parameters that yielded the lowest score, see Formula 3.1.

Algorithm configuration	Parameter	Label	Value	Score
Baseline	Protocol period (ms)	T_b	1000	0.70835
	Suspicion duration (ms)	S_b	30000	
Optimal Lighthouse	Protocol period (ms)	T_l	1000	0.285499
	Suspicion duration (ms)	S_l	10000	
Optimal SWIM	Round trip time (ms)	RTT	1000	0.585061
	Nodes to probe	k	2	
	Lambda	λ	10	

The scores for each of the six key evaluation metrics that were obtained by Baseline, Optimal Lighthouse, and Optimal SWIM are presented in Table 5.2. Two noteworthy observations are that all three algorithm configurations detected all failures and that only Baseline achieved a 0% false positive rate.

Table 5.2: Average scores for each key metric based on three simulations for each optimal configuration.

		Algorithm configuration		
Evaluation Category	Evaluation Metric	Baseline	Optimal Lighthouse	Optimal SWIM
Speed	Average Full Dissemination (ms)	32455	13984	32292
	Average First Detection (ms)	30297	11627	22265
Accuracy	Undetected Failure Rate	0.0	0.0	0.0
	False Positive Rate	0.0	5.0	15.4
Network Load	Total Messages Sent	363	362	252
	Message Load (Bps)	52245	53934	40018

While Table 5.2 displays the actual values for each of the six key metrics, Figure 5.2 displays Optimal Lighthouse and Optimal SWIM's relative performance compared to Baseline. Two highlights from Figure 5.2 are that both Optimal SWIM and Optimal Lighthouse outperform Baseline in nearly all metrics. Noteworthy achievements are that Optimal Lighthouse is nearly 57% faster at disseminating information than both Optimal SWIM and Baseline. Furthermore, Optimal Lighthouse is 61.6% faster at detecting failures than Baseline while Optimal SWIM is only 26.5% faster. For the two metrics related to network load, total messages sent, and message load, Optimal Lighthouse is more or less achieving the same results as Baseline. Meanwhile, Optimal SWIM sends 30.7% less messages than Baseline while reducing the bytes sent per second by 23.4%.

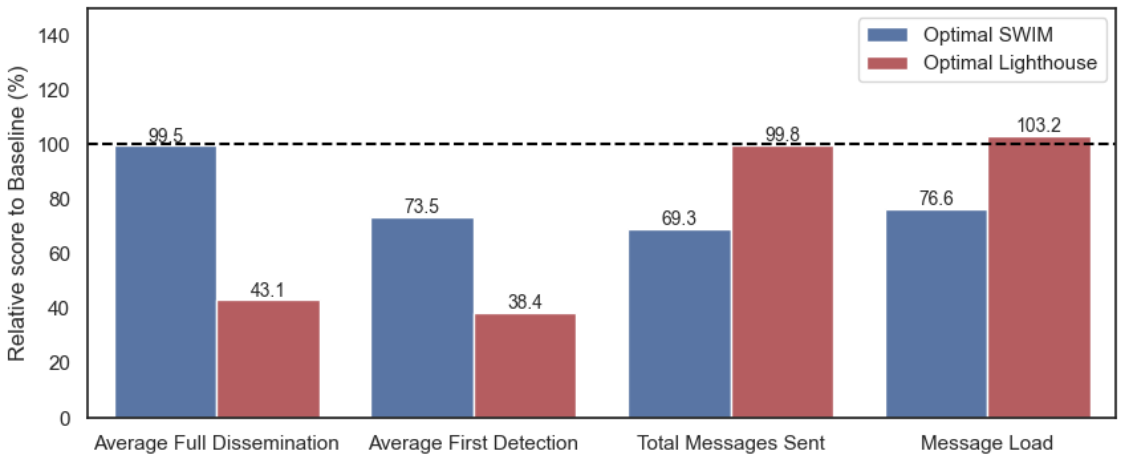
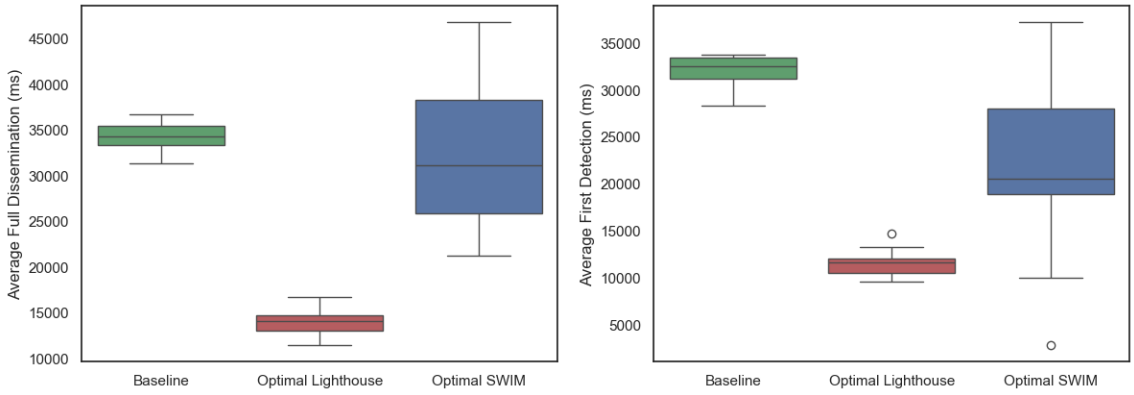


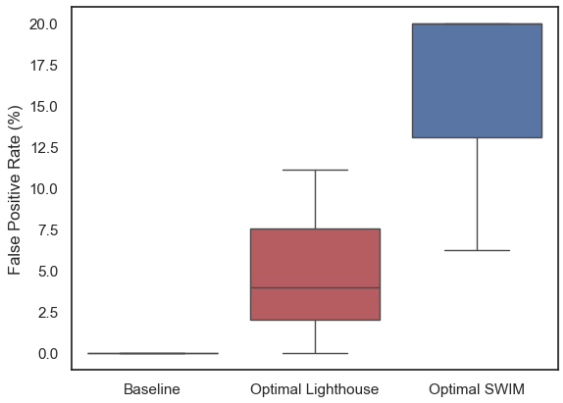
Figure 5.2: Relative performance of Optimal Lighthouse and Optimal SWIM compared to Baseline which is represented by the dotted line. The result is the average score from three simulations for each algorithm configuration.

When running three experiments for each optimal algorithm configuration, it is evident that SWIM's performance varies more than its competition as a wider interquartile range (IQR) is observed in five out of six metrics, see Figure 5.3. More specifically, the metrics that vary the most are average full dissemination, average first detection, and false positive rate, see Figure 5.3a, 5.3b, and 5.3c.

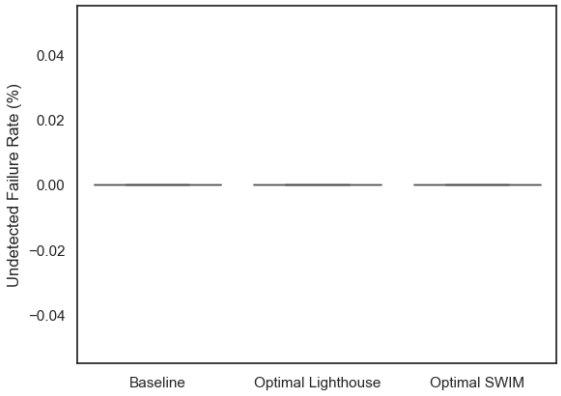


(a) Average Full Dissemination (ms)

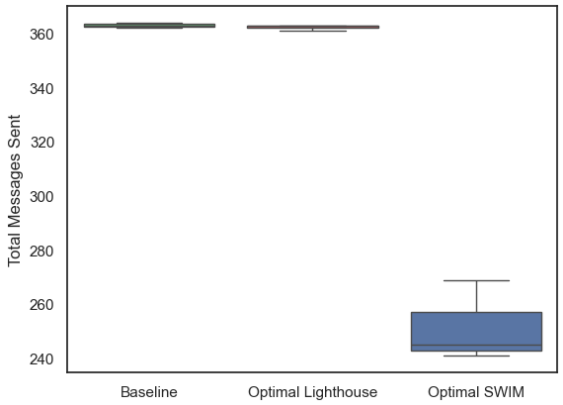
(b) Average First Detection (ms)



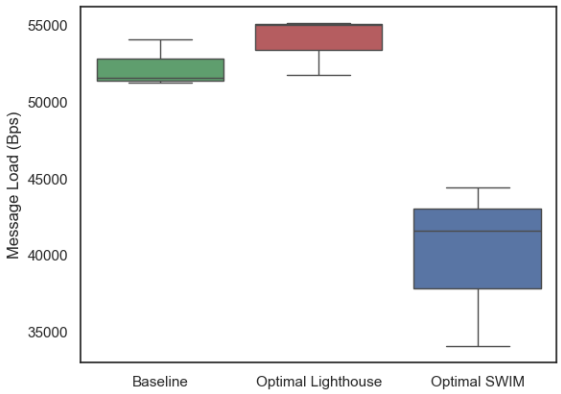
(c) False Positive Rate (%)



(d) Undetected Failure Rate (%)



(e) Total Messages Sent



(f) Message Load (Bps)

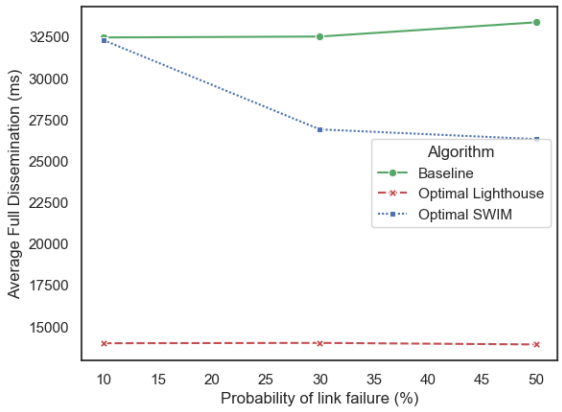
Figure 5.3: Variation in result for Baseline, Optimal Lighthouse, and Optimal SWIM. The result is the average score from three simulations for each algorithm configuration.

5.2 Further Investigations

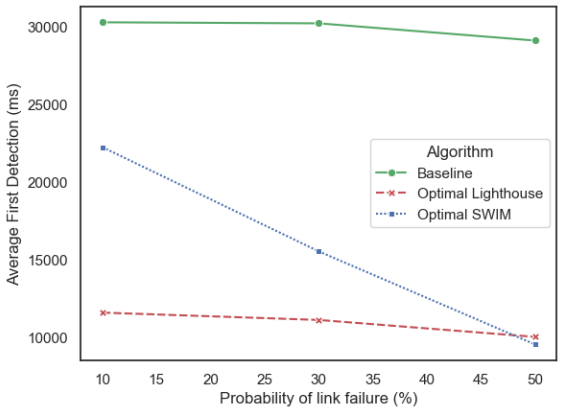
This section will provide further investigations of the previously identified optimal configurations in regard to link failures and scalability.

5.2.1 Link Failure

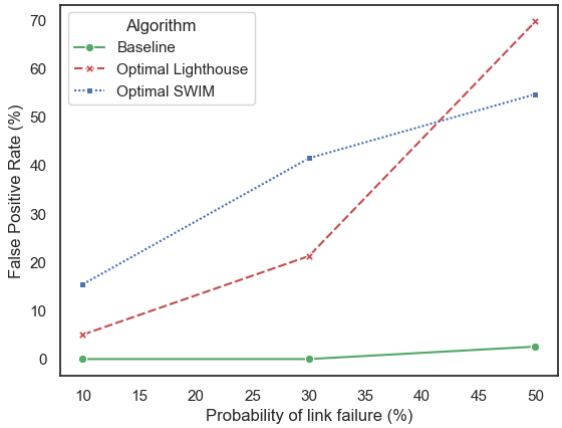
The result from exposing the three algorithm configurations to a network more prone to link failures is displayed in Figure 5.4. It shows that the average full dissemination speed is stable for both Baseline and Optimal Lighthouse when increasing the number of link failures. Meanwhile, Optimal SWIM is faster at disseminating information, see Figure 5.4a. Similarly, both Baseline and Optimal Lighthouse show a stable, but slight decrease, in their average first detection time while Optimal SWIM is becoming a lot faster, see Figure 5.4b.



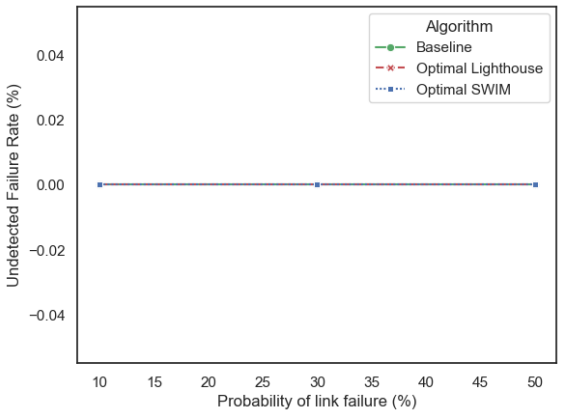
(a) Average Full Dissemination (ms)



(b) Average First Detection (ms)



(c) False Positive Rate (%)



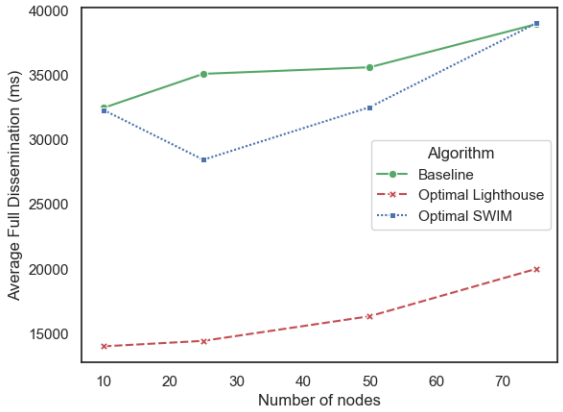
(d) Undetected Failure Rate (%)

Figure 5.4: Baseline, Optimal Lighthouse and Optimal SWIM's ability to handle networks more prone to link failures. The result is the average score from three simulations for each algorithm.

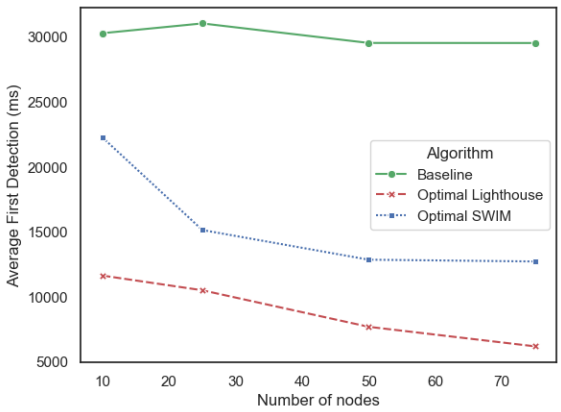
Furthermore, when increasing the probability of link failure, undetected failures remain at 0%, see Figure 5.4c. However, the rate of false positives is increasing rapidly for both Optimal Lighthouse and Optimal SWIM while Baseline remains stable, see Figure 5.4d.

5.2.2 Scalability

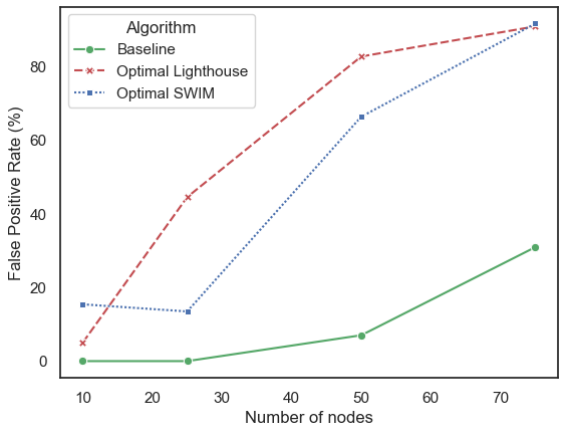
Figure 5.5 visualises the three algorithm configurations' scalability potential as the cluster size varies from 10 to 75 nodes. For most cases, the three algorithm configurations show similar trends but there are a couple of observations that deserve to be highlighted. Firstly, the dissemination speed is slower for all three configurations when increasing the cluster size, see Figure 5.5a. Secondly, the average first detection speed is faster for both Optimal Lighthouse and Optimal SWIM while it remains similar for Baseline, see Figure 5.5b. Thirdly, Optimal SWIM and Optimal Lighthouse are a lot more sensitive to larger cluster sizes as the number of false positives increases rapidly up to over 80%, see Figure 5.5c. Fourth, the first, and only, undetected failures are reported by Optimal SWIM when there are over 25 nodes, see Figure 5.5d. Fifth, both Baseline and Optimal Lighthouse show a tremendous increase of Bytes sent per second when the size of the cluster increases. Meanwhile, Optimal SWIM is relatively stable as it does not increase at the same pace, see Figure 5.5f.



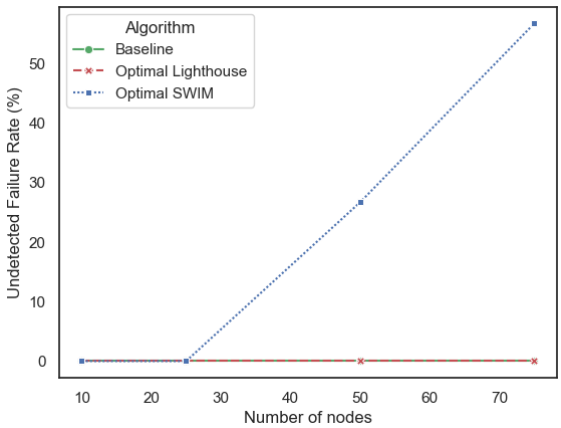
(a) Average Full Dissemination (ms)



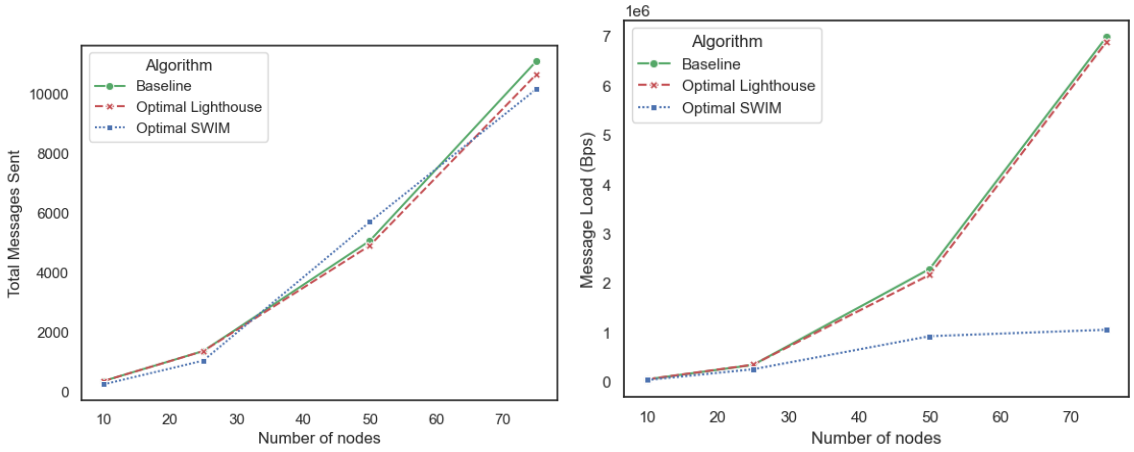
(b) Average First Detection (ms)



(c) False Positive Rate (%)



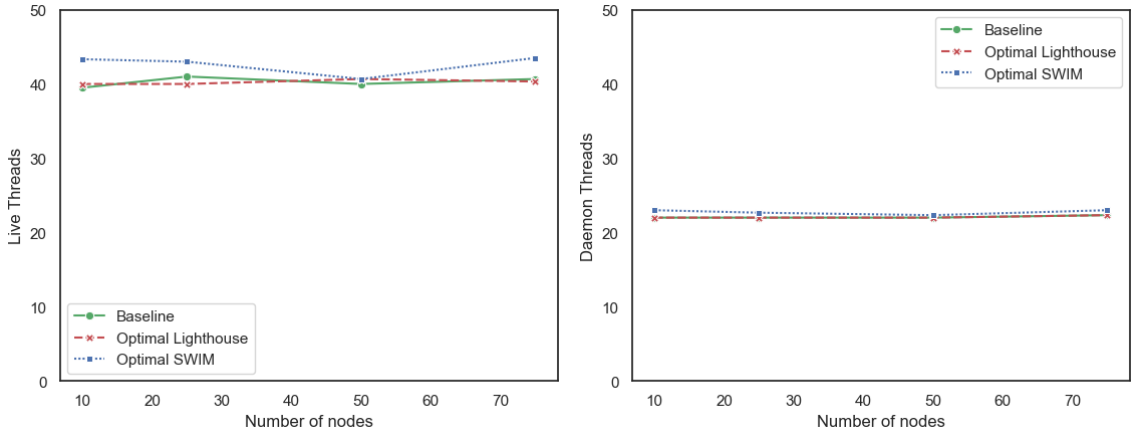
(d) Undetected Failure Rate (%)



(e) Total Messages Sent **(f)** Message Load (Bps)

Figure 5.5: Scalability potential for Baseline, Optimal Lighthouse, and Optimal SWIM by varying the number of nodes in the cluster from 10 to 75. The result is the average score from three simulations for each algorithm configuration.

Lastly, while running experiments on scalability, the number of threads that were used by arbitrary nodes was observed, see Figure 5.6. Note that there is not much variation in the number of active or daemon threads as the number of nodes increases for any algorithm.



(a) Live Threads **(b)** Daemon Threads

Figure 5.6: Active threads in one randomly selected node for each algorithm configuration while varying the number of nodes in the cluster. The number of threads is the average count from three simulations for each algorithm configuration.

Chapter 6
Discussion

This chapter will discuss the results in a bigger context by explaining the impact of the underlying design decisions and threats to their validity.

6.1 Analysis of Results

This section will discuss four aspects of the result: parameter optimisation, variation, link failures, and scalability.

6.1.1 Parameter Optimisation

Even though parameter tuning shows great potential to increase speed while maintaining reasonable accuracy, it has a couple of pitfalls that should not be neglected. Firstly, performing parameter optimisation is an extremely time-consuming and computationally heavy process. In our test environment, a single simulation took roughly 5 minutes. As a result, for the parameter optimisation, we only ran three simulations for each configuration which ultimately resulted in 99 experiments that took nearly 8 hours.

Secondly, due to the time-consuming nature of the parameter-tuning process, we had to reduce the number of tested parameters for both the algorithms and the experimental setup. For example, we only simulated a cluster with 10 nodes and a network with a 10% probability of link failure. Limiting the scope of the parameter tuning turned out to have a larger impact than expected as we accidentally ended up tuning the algorithms too hard on the specific environment. On one hand, Optimal Lighthouse and Optimal SWIM initially showed great potential to replace Baseline as they outperformed it in all metrics except message load, see Figure 5.2 and Table 5.2. However, when varying the size of the cluster and the probability of link failure, there is a completely different story as Baseline is a lot more stable, see Figure 5.5 and 5.4. In conclusion, while parameter tuning shows great potential, it is crucial to tune the system in an experimental setup as close to the real world as possible.

6.1.2 Variation

In regards to variation, it is evident that Baseline is the most stable configuration as it shows a low variety in results, see Figure 5.3. This section will discuss three reasons why some algorithms generate more variety than others to better understand the algorithm's strengths and weaknesses.

Firstly, Optimal Lighthouse varies about as little as Baseline while Optimal SWIM is varying a lot for the two-speed metrics, see Figure 5.3a and 5.3b. One explanation for this behavior is that SWIM either succeeds on the direct ping or it chooses to probe k members for help. Meanwhile, Optimal Lighthouse and Baseline only directly ping another member, i.e. it does not ask anyone for help if the ping fails. Therefore, SWIM has two alternative outcomes while Baseline and Optimal Lighthouse only have one which ultimately leads to more variation for SWIM.

Secondly, both Optimal SWIM and Optimal Lighthouse produce false positives while Baseline does not, see Figure 5.3c. One of the reasons for this is arguably, as described in Section 6.1.1, that both Optimal SWIM and Optimal Lighthouse was tuned too hard in the parameter optimisation which causes them to be more aggressive.

Lastly, Optimal SWIM sends a lot fewer messages and bytes per second compared to Baseline and Optimal Lighthouse, see Figure 5.3e and 5.3f. However, even though Optimal SWIM puts less pressure on the network, its pressure still varies more than its competitors. One of the explanations why Optimal SWIM sends fewer messages per second is that Optimal SWIM only sends the delta, i.e. only the identified changes in membership state, while Baseline and Optimal Lighthouse always send the full state. That means that Optimal SWIM will send more data when a change has occurred compared to when nothing has changed which leads to more variety.

6.1.3 Link Failure

Optimal SWIM shows promising results in a network that is more prone to link failures, see Figure 5.4. First of all, it is once again important to note that both Optimal SWIM and Optimal Lighthouse had a narrow suspicion duration which ultimately makes them fast, but have some false positives, see Figure 5.4c. Furthermore, one of the reasons why SWIM is better in a network more prone to link failures is that it probes k members, i.e. asks neighbouring nodes for help if the first direct ping fails. Meanwhile, Optimal Lighthouse and Baseline will suspect the node on the other side of the failed link, and there is a race against suspicion duration if the suspected node is able to refute the suspicion. This trend is visible in Figure 5.4c as Optimal SWIM's false positive rate is not increasing as rapidly as Optimal Lighthouse.

6.1.4 Scalability

As Neo4j aims to support larger cluster sizes in the future it is important to consider how well each algorithm scales as the number of nodes increases. All algorithms have a decreased average full dissemination speed when increasing the cluster size. This trend is reasonable as there are more nodes that have to receive the update and more pings that have to be made to propagate the information across the cluster, see Figure 5.5a.

One observation in Figure 5.5b is the fact that the first detection speed for the optimised algorithms decreases below its suspicion duration while Baseline's detection time remains somewhat constant. The suspicion duration is 10s for Optimal Lighthouse and 20s for Optimal SWIM. This happens due to the fact a node is suspected before the crash actually happened and since it died it will not oppose the suspicion, as it would do in the case that the node was alive. This contributes to the fact that the suspicion duration is started before the crash occurred and better detection time is observed. The situation becomes more apparent when the number of nodes increases and more pings are made by more nodes. To summarise, the algorithms' detection speed does not become worse when varying the cluster size and they can therefore be considered scalable.

The parameter optimisation used two metrics, first detection speed, and false positives rate, to determine the optimal parameters for each algorithm. The first detection speed of the algorithms was shown to be scalable. However, the false positives rate in Figure 5.5c presents another perspective on whether or not the chosen parameters are scalable. The false positive rate increases from 0% to above 20% for Baseline when increasing the cluster size. But, for the optimised algorithms the false positive rate ends with a value above 80%. A false positive rate above 80% is unacceptable in this context, where a majority of the determined kills are incorrect. That shows that the optimised algorithms do not scale well regarding the false positive rate. This is due to the fact that the suspicion duration of the Optimal SWIM and Optimal Lighthouse is too narrow. Earlier on in the section, it was concluded that the dissemination speed became slower while increasing the cluster size, i.e. that it takes a longer time to propagate information around the cluster. It results in that it takes a longer time for a node to receive the suspected state, oppose it, and propagate its alive state back to the node that was originally suspecting it. This behavior is observed when comparing Optimal Lighthouse and Baseline, and the only difference between them is the suspicion duration. With Baseline's suspicion duration, it has a false positive rate above 20% when cluster size is 75 and Optimal Lighthouse has a rate above 80%. Nodes have a chance to refute the suspicion with Baseline and not with Optimal Lighthouse.

Considering the network load, it is reasonable that the total messages sent increase with the cluster size, as Figure 5.5e shows. All the algorithms roughly send the same number of messages, even with the fact that Optimal SWIM has a longer protocol period, which means that Optimal SWIM makes fewer direct pings per second to the selected node. Since Optimal SWIM sends an indirect ping to two other nodes and they in turn send ping requests, the number of messages sent by the SWIM algorithm increases.

Even though the number of messages sent by the algorithms is the same, there is a big difference in the message load between Optimal SWIM and the Lighthouse-based algorithms when varying the cluster size, see Figure 5.5f. Baseline and Optimal Lighthouse's trend is an exponential increase with the cluster size and on the contrary for Optimal SWIM, the trend is more a linear increase. Since the number of sent messages is approximately the same between the algorithms, the only thing that can differ is how much data is sent. The difference between the SWIM and Lighthouse algorithms regarding what is data sent is that Lighthouse sends the full state list, the length of it increases with the cluster size, and SWIM only propagates update messages around the cluster. This means that SWIM shows great potential for scalability regarding message load.

To summarize, all algorithms show scalability when considering full dissemination and first detection speed. However, regarding the false positive rate, Baseline is the only algo-

rithm that has a reasonable rate, the other ones have too narrow suspicion duration which contributes to a large false positive rate. Looking at the network load, Optimal SWIM is the only algorithm whose message load does not increase exponentially. Since the optimised algorithms' rate of false positives is insufficient, Baseline remains the only choice when considering the best algorithm that is scalable.

As seen in Figure 5.5d, SWIM reports a lot of undetected failures when the cluster size increases. This is alarming since it is does not satisfy the requirements of a scalable failure detector. On the other hand, Neo4j does not use larger clusters at the moment so it is currently not a problem, but it will be in the future. One possible hypothesis is that it was not run long enough so that the failures are detected. However, due to a lack of time and granular logs, we have not yet been able to identify what causes this behaviour and we leave this for future work.

6.2 Limitations and Threats to Validity

When evaluating algorithms it is important to identify threats that can affect the validity of the results. During the evaluation of the algorithms, we identified three threats.

The first threat of the results is that the parameter tuning was made too narrow, i.e. too few combinations were tested. As a result, the obtained results may not portray the algorithms accurately as the chosen parameters may not have been realistic enough.

Secondly, due to time-consuming and computationally heavy simulations, the test had to be relatively short which ultimately made the results more vulnerable to outliers. For example, the false positive rate in a small test may appear larger than it actually is as it is heavily dependent on the random seed. Let's say that we detect 4 crashes in a small test and 1 is a false positive. Then we get a 25% false positive rate. Meanwhile, this may just have been unlucky as we might also just have gotten 1 false positive in a test with 100 crashes. However, it is also important to note that perhaps it could have been the other way - perhaps we were lucky. We tried to mitigate this by running three simulations for each test but we were limited due to time.

Lastly, while running the simulation, each node had roughly 40 active threads regardless of the cluster size, see Figure 5.6. When increasing the cluster size to 75 nodes this results in 3 000 threads in total. The effect of this is that there will be a lot of context switches between threads since the computer used for evaluation cannot run all those threads concurrently. A computer can run a specified number of threads concurrently and when the number of threads exceeds that number, the computer switches between the threads on which one to execute. It may appear that the threads are running concurrently but in reality, they are to some extent running sequentially. This behavior may result in delays and timing issues as the threads that need to refute suspicion allegations could be idle. This is why it would be a wise decision to migrate the simulation engine to the cloud where each node could be run on a separate server.

Chapter 7
Conclusion

This chapter will present our conclusions to the research questions and provide ideas on possible future work.

7.1 Research Questions

RQ 1: What Are the Strengths and Weaknesses of Baseline?

Baseline, i.e. the Lighthouse failure detector with the initial parameters, shows great stability and high accuracy in clusters with 10 to 75 nodes. However, the parameters have not yet been optimised for Neo4j as they are solely based on the pre-defined parameters in Akka. Therefore, it is arguably slow due to the long suspicion duration of 30s.

RQ 2: Which Alternative Algorithms Exist and How Do They Compare to Baseline?

After researching related work it was discovered that well-established vendors in the industry leverage the SWIM algorithm and the φ failure detector. Neo4j is also using SWIM as Lighthouse is a modification of it. Compared to Baseline, the optimised SWIM and Lighthouse reduced the detection and dissemination speed while maintaining a reasonable accuracy for clusters with 10 nodes. Furthermore, SWIM generates a considerably lower network load compared to its competitors. While investigating the scalability potential it was evident that both Optimal Lighthouse and Optimal SWIM were tuned too hard which gave an insufficient false positive rate. Therefore, Baseline could be considered a conservative choice while both Lighthouse and SWIM are more aggressive.

RQ 3: How Can Neo4j Improve Their Usage of Failure Detection Algorithms?

While the Lighthouse-based algorithms show potential of great speed and accuracy, they generate a lot of network load. Therefore, a simple improvement can be to propagate state updates rather than the full membership list similar to SWIM. Meanwhile, there are two more demanding ways that Neo4j can improve its use of failure detectors: by finding each algorithm's optimal parameters and by developing a more advanced evaluation engine. Firstly, it shows that parameter tuning has great potential to improve a failure detector's performance. Perhaps even better potential than the choice of algorithm. Secondly, to find optimal parameters, a testing environment that allows more systematic and consistent real-world simulations is required.

7.2 Future Work

Due to the limited scope of this thesis, there are five additional ideas that we identified to be interesting topics for future work as we did not have time to complete them.

Firstly, migrate the simulation engine to the cloud and run each node on its own server. Not only would this more accurately simulate a real network, but it would also allow more comprehensive testing and parameter tuning as simulations could be run over a longer time in the background. As a result, outliers could potentially be mitigated as their impact on the final result is minimized when more tests are conducted.

Secondly, investigate how more input could be added to the parameter optimisation process to be able to find the optimal configurations that are performing better on more aspects, not only first detection speed and false positive rate. For example, by considering more metrics and several experimental setups, e.g. varying latency and cluster sizes, in order to find parameters that perform well in different environments.

Thirdly, while Optimal SWIM shows great potential, it still lags behind Optimal Lighthouse. However, SWIM could be improved further which may make it even better than Optimal Lighthouse. For example, SWIM can be less dependent on its defined parameters and more adaptable to varying networks by implementing the Lifeguard extension. More specifically, Lifeguard adapts its timeouts during the execution based on the node's health.

Further on, to provide a more accurate experiment setup a further improvement would be to monitor Neo4j's real networks to create an experimental setup that is more similar to the real world. E.g. by measuring the actual number of lost packets, round-trip time, and the usual number of nodes that fail. In the current approach, the parameters of the experimental setup were solely based on discussions with the team and not a decision based on data. A more realistic experimental setup would contribute to a more accurate evaluation of the failure detection algorithm.

Lastly, Neo4j has the ambition to support clusters with up to 100 nodes and their customers may want clusters with varying sizes, from 5 to 100 nodes. Hence, it would be interesting to further research how to dynamically adjust the algorithms to the cluster size to avoid the need for tuning parameters for different deployments.

References

[1] Forbes. Internet usage statistics in 2024. Available At: https://www.forbes.com/home-improvement/internet/internet-statistics/#:~:text=There%20are%205.35%20billion%20internet%20users%20worldwide.&text=Out%20of%20the%20nearly%208,the%20internet%2C%20according%20to%20Statista. Accessed: 2024-04-04.

[2] Geeksforgeeks. What is a distributed system? Available At: https://www.geeksforgeeks.org/what-is-a-distributed-system/. Accessed: 2024-02-15.

[3] Bhavana Chaurasia and Anshul Verma. A comprehensive study on failure detectors of distributed systems. *Journal of Scientific Research*, 64(2):250–260, 2020.

[4] Neo4j. Neo4j named a visionary in 2023 gartner® magic quadrant™ for cloud database management systems. Available At: https://neo4j.com/blog/gartner-magic-quadrant/. Accessed: 2024-02-13.

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

[6] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, pages 303–312. IEEE, 2002.

[7] Armon Dadgar, James Phillips, and Jon Currey. Lifeguard: Local health awareness for more accurate failure detection. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 22–25. IEEE, 2018.

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

[9] IBM. What is a relational database? Available At: https://www.ibm.com/topics/relational-databases#:~:text=A%20relational%20database%20is%20a,key%20or%20a%20foreign%20key. Accessed: 2024-04-04.

[10] Neo4j. What is a graph database? Available At: `https://neo4j.com/docs/getting-started/get-started-with-neo4j/graph-database/`. Accessed: 2024-04-04.

[11] Maarten Van Steen and Andrew S Tanenbaum. *Distributed Systems*. Createspace Independent Publishing Platform, North Charleston, SC, 3 edition, February 2017.

[12] Jean Dollimore George Coulouris and Tim Kindberg. *Distributed Systems*. International Computer Science Series. Addison-Wesley Educational, Boston, MA, 4 edition, June 2005.

[13] Diego Ongaro. *Consensus: Bridging theory and practice*. Stanford University, 2014.

[14] Martin Kleppmann. *Designing data-intensive applications*. O'Reilly Media, Sebastopol, CA, March 2017.

[15] Michael K Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996.

[16] João Leitão, José Pereira, and Luís Rodrigues. Gossip-based broadcast. *Handbook of Peer-to-Peer Networking*, pages 831–860, 2010.

[17] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 350–361, 2011.

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

[19] Akka. Cluster specification. Available At: `https://doc.akka.io/docs/akka/current/typed/cluster-concepts.html#failure-detector`. Accessed: 2024-04-12.

[20] Indranil Gupta, Tushar D Chandra, and Germán S Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 170–179, 2001.

[21] Toxiproxy. Readme. Available At: `https://github.com/Shopify/toxiproxy/tree/main`, Last updated: 2023-04-18. Accessed: 2024-04-18.

[22] Prometheus. Overview. Available At: `https://prometheus.io/docs/introduction/overview/`. Accessed: 2024-04-18.

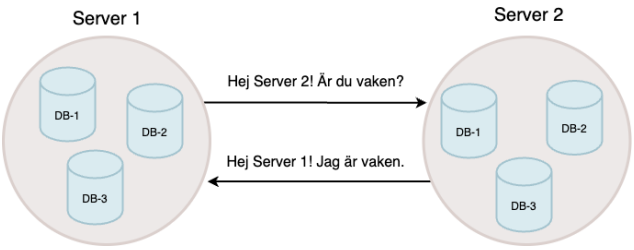
EXAMENSARBETE Navigating Failures in Distributed Systems:
A Comparative Study of Failure Detection Algorithms
STUDENTER Hannes Brinklert, Johan Åkerman
HANDLEDARE Jonas Skeppstedt (LTH), Aleksey Karasavov (Neo4j)
EXAMINATOR Michael Doggett (LTH)

Hur hittas fel i distribuerade system?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Hannes Brinklert, Johan Åkerman**

Distribuerade system används idag allt mer för att köra applikationer som ställer högre krav på prestanda och pålitlighet. Denna studie har undersökt hur Neo4js världsledande grafdatabaser kan bli mer pålitliga med hjälp av feldetetekteringsalgoritmer.

Distribuerade system består av geografiskt åtskilda noder som är ihopkopplade med länkar över ett nätverk. Vad noderna i fråga faktiskt är varierar på användningsområdet. I Neo4js fall är varje nod en server som kör en eller flera databaser. En av de större ingenjörsutmaningarna med distribuerade system är samarbete då alla noderna har sin egen syn på systemet som en helhet. Feldetekteringsalgoritmer används för att lösa detta problem genom att hitta noder som inte fungerar som de ska, se exempel nedan.



I detta examensarbete har vi under fem månader samarbetat med företaget Neo4j som ligger i framkant inom grafdatabaser. Mer specifikt så har vi undersökt två alternativa vägar framåt för att förbättra Neo4js feldetektorer som i sin tur ökar pålitligheten av databaserna som används av 75% av Fortune 100 företag. Det första alternativet bestod av att optimera valet av parametrar i Lighthouse, företagets redan existerande algoritm. Det andra alternativet var att ersätta Lighthouse med en ny algoritm med namn SWIM. Den största skillnaden mellan Lighthouse och SWIM är att i SWIM tar en nod hjälp av sina grannar och skickar mindre och mer relevanta meddelanden.

Den ursprungliga Lighthouse-algoritmen användes som en referenspunkt och jämfördes mot en parameteroptimerad version av både Lighthouse och SWIM. De tre konfigurationerna utvärderades i ett simulerat nätverk som utsattes för tre olika typer av fel: krasch, länk och fördröjningsfel.

Under simuleringen utvärderades de tre algoritmerna utifrån tre perspektiv: snabbhet, träffsäkerhet och belastning på nätverket. Resultatet visar att den ursprungliga Lighthouse-algoritmen är skalbar och pålitlig men långsam. Jämfört med den etablerade referenspunkten så är den parameteroptimerade Lighthouse-algoritmen 61.6% snabbare på att hitta fel men något mindre träffsäker (5%). Samtidigt är den nya SWIM-algoritmen något snabbare än referenspunkten (26.5%) men betydligt mindre träffsäker (15.4%). En viktig observation är att SWIM utsätter nätverket för betydligt mycket mindre trafik vilket blir särskilt uppenbart i distribuerade system med fler noder. Den något överraskande slutsatsen som kunde dras från resultatet är att optimering av algoritmernas parametrar har större påverkan än själva valet av algoritm.