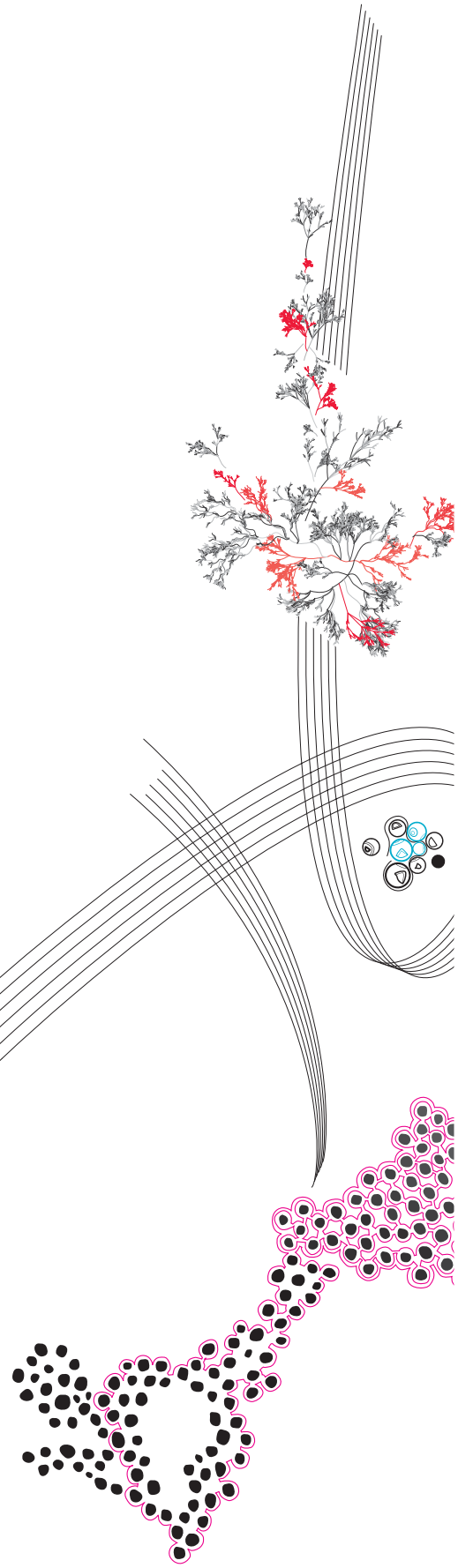

MSc Computer Science
Final Project

Improving the Failure Detection Time of Distributed Mission-Critical Systems

Aleksandra Siderova

Supervisors: Luís Ferreira Pires,
Tiago Prince Sales, & Willy Boenink

April, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

In this thesis, the problem of fast failover in a distributed mission-critical system using Kubernetes is explored. The thesis aims to provide a comprehensive background on the problem, as well as to compare potential solutions that could address the challenges at hand. The technologies considered are Serf, Consul and ZooKeeper, with Serf being selected as a suitable candidate for comparison in an experimental environment that simulates the conditions of a distributed mission-critical system. The findings of this thesis indicate that Serf offers an improved failure detection time compared to the use of Kubernetes alone. Furthermore, a range for the expected failure detection time is established, although its results may not be sufficiently predictable for the stringent requirements of mission-critical systems.

Keywords: Kubernetes, health monitoring, failover, failure detection, mission-critical systems, distributed systems, Serf, DDS

Chapter 1

Introduction

In this chapter, the motivation for this project and its corresponding research questions are established, as well as the approach taken to answer them and the overall structure of the thesis.

1.1 Motivation

The rapid growth of modern IT systems has given rise to a wide variety of challenges that require suitable architectural solutions to tackle them. The more a system increases in size, the more complex it is to manage it, since maintenance, debugging and deployment of the system becomes more difficult and potentially less efficient. In addition, any changes made require longer downtimes, flexibility is highly limited, and scaling the system to support the growing demand is inefficient. These are only some of the problems organizations can encounter when dealing with growing, complex systems [24]. Furthermore, due to the fast pace at which system architectures need to change in order to meet critical safety and security requirements, adaptability is a necessary non-functional requirement that needs to be met [15]. As such, a robust and efficient solution that can adapt to conditions specific to the mission-critical domain is needed.

Cloud architectures are a modern solution that addresses the need for scalability, resilience and adaptability in software, by enabling the on-demand distributed use of resources over a network [2]. As a result, cloud architectures have become widely adopted across different industries. For organizations dealing with mission-critical systems, the preferred type of setup is often on-premises due to the need to prioritize security [2], as outsourcing the infrastructure inherently brings security risks. Additionally, the architecture of such systems can differ from the typical cloud architecture, such as having fewer nodes and a smaller scale, for example in systems within the naval, aerospace and defense industries. Furthermore, for mission-critical systems where data exchange should be reliable and efficient, middleware for secure, high-performance communication between the various components in the system, such as the Data Distribution Service (DDS), is required [39].

In cloud architectures, containerization is a commonly used technology [14]. Containerization is a type of virtualization that involves packaging an application with all of the libraries, configuration files, dependencies, etc. that are required for a successful deployment. This packaging allows containers to run on any platform for enhanced portability [10] with minimal overhead, which has resulted in the rapid adoption of containerization

in various industries [54].

The de facto standard for managing containers is Kubernetes [20], which is an open-source orchestration platform for cluster management. Kubernetes allows users to request resources in an abstract manner, while the platform handles the resource allocation and scheduling, leading to more convenient development and management of applications [14]. Furthermore, Kubernetes provides a multitude of capabilities that contribute to improved scaling, portability, automated deployment, resource management, health monitoring and recovery, and more [8]. The flexibility and extensive solutions that Kubernetes offers have led to its rapid industry-wide adoption, with approximately 70% of IT leaders worldwide working for organizations that utilize Kubernetes in their processes, as surveyed by RedHat [22].

Although mission-critical systems can benefit from using Kubernetes to address the aforementioned challenges, a specific requirement that they must also fulfill is that they should guarantee high availability and reliability of the systems. In some domains such as defense, healthcare, etc., significant downtime can cause hazards such as property damage, environmental damage, and even harm to human life. As such, any solution used in a mission-critical environment must also be able to support near-uninterrupted availability.

Despite its many benefits, Kubernetes alone is not always enough to meet the requirements of systems with stringent availability targets [5], including mission-critical ones, where it is crucial to be able to detect and recover from failure in as short a time as possible under conditions specific to the domain. While Kubernetes offers support for failure detection and recovery, on its own it cannot always do so efficiently enough to be suitable for mission-critical systems, where recovery needs to occur near-instantaneously, since the failure detection phase alone can take up to 40 seconds when using Kubernetes [3]. As such, it is necessary to explore an approach that combines the features of Kubernetes which are suitable for mission-critical systems, such as scheduling, automatic scaling, networking, etc., with a more efficient way of keeping track of and communicating the state of the system, as well as triggering the recovery process in the event of failure.

1.2 Research Questions

This thesis aims to investigate how to improve the failure detection time of a mission-critical distributed system managed by Kubernetes that communicates via DDS. The main problem can be addressed by defining the following research questions:

RQ1. Which technology could be a suitable solution for the problem?

a. What technologies can improve the failure detection time of a distributed system managed by Kubernetes?

b. What are the advantages and disadvantages of each of these technologies?

c. How can each of these technologies be used in combination with Kubernetes?

RQ2. What are the practical implications of using the selected technology to address the established problem?

a. What are the key features of the selected technology that contribute to failure detection and communication in distributed systems?

b. How does the selected technology perform in an experimental setting compared to the capabilities of Kubernetes alone?

c. What impact does increasing the number of physical nodes have on the system?

1.3 Approach

In this section, the steps taken in order to answer the research questions posed in Section 1.2 are described in detail.

Given that this research aims to present an improvement to the existing solution of an organization via the delivery of a software artifact, *Design Science* [36] is a suitable research methodology to follow. In particular, Design Science describes the iterative creation of an information system via the switching perspectives of *processes* and evolving *artifacts*, i.e. products. The use of Design Science allows for a better understanding of the established problem, and further provides a way to assess the feasibility of the selected approach implemented by the artifact. Afterwards, the artifact is improved accordingly throughout the next iteration until a satisfactory result has been reached.

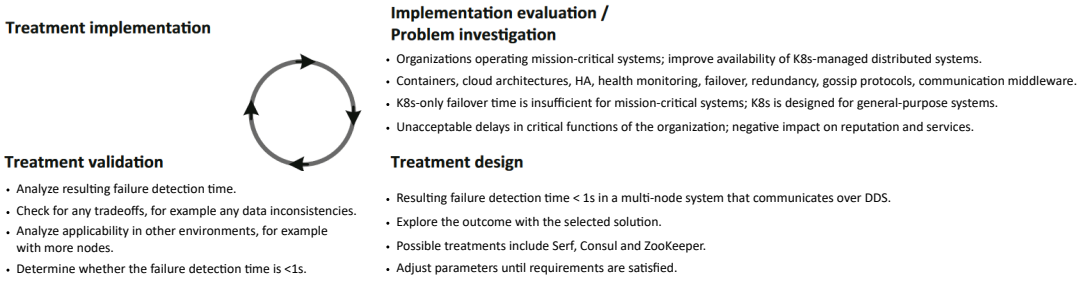

FIGURE 1.1: Wieringa, R. (2014) [59]. *The engineering cycle.* Adapted to suit the process of this project.

The overall cycle is depicted in Figure 1.1 following the template in [59], with the steps taken to answer the research questions. The problem investigation phase of the cycle was used to answer RQ1, the approach for which is described in more detail in Section 3.1, while the subsequent stages and their iterations focused on answering RQ2.

In particular, the following five stages of the process can be identified:

1. Literature review - primarily used to discover potential solutions to the problem, as well as to provide background knowledge to understand the problem context (Section 2). Furthermore, important steps such as identifying stakeholders and requirements were also taken during this stage, as described later in this section.

2. Comparison of potential solutions - after the potential solutions have been identified, a comparison is made between them in order to decide which is most suitable to proceed with (Chapter 3).

3. Implementation - once a solution has been selected, a mock-up of the system that meets the identified prerequisites was implemented as preparation for the experimental phase (described in Section 4.3).

4. Optimization - the implementation derived from the previous stage was optimized

until the desired results were met, following an examination of the relevant mechanisms (described in Section 4.2).

5. Experiments - the final stage involved collecting data from the experiments and discussing the findings (Section 4.4).

More specifically, the following stakeholders were identified:

- Organizations that deal with distributed mission-critical systems.
- Organizations that incorporated Kubernetes in their systems.
- System architects and engineers.
- Cloud infrastructure providers.
- Regulatory agencies.

The following requirements were taken into consideration during the implementation:

- A Kubernetes-only mock-up must be implemented.
- A mock-up that integrates the selected solution within Kubernetes must be implemented.
- The two implementations must be optimized.
- The failure detection time of the two implementations must be compared, both with default and optimized settings.
- Failure detection time must be less than 1 second for the selected solution after optimization.
- The system must use at least 2 physical nodes.
- Communication between the nodes must be done via data-centric middleware relevant to mission-critical systems, such as DDS.
- The Kubernetes-only implementation must incorporate failover via cold standby.
- The implementation that integrates the selected solution must incorporate failover via warm standby.

In order to meet these requirements and successfully answer the established research questions, a mock-up was implemented that simulates multiple physical nodes communicating with each other over DDS. A node failure was then introduced and the failure detection time was recorded across multiple experiments, in particular one with Kubernetes only and one with the integrated selected technology. Furthermore, a comparison is also made between the default and optimized settings of each implementation.

1.4 Project Collaboration

This research is done in collaboration with *Thales*, an organization involved in the defense and aerospace sector. More specifically, the implementation of this project follows the guidelines and expertise derived from Thales's experience with building a failure detection mechanism for their naval systems in the event that a node located on a ship malfunctions. As such, the requirements and technologies utilized in the implementation follow Thales's expectations and guidelines.

At the moment, Thales's current solution involves each node sending a heartbeat, and if two consecutive heartbeats are missed, the node is considered dead. Their system achieves a failure detection time of approximately 600 milliseconds to 1 second. Their system is also legacy software with unknown functionalities and uncertain behaviour, all of which are aspects that this research aims to improve upon.

1.5 Thesis Structure

The remainder of the thesis is organized as follows. Chapter 2 provides the background knowledge necessary to understand the work done throughout this thesis, Chapter 3 provides a comparison between the candidates for improved failover, Chapter 4 outlines the architecture of the experiments, an in-depth analysis on the selected technology, and a discussion of the obtained results, and Chapter 5 concludes the thesis.

Chapter 2

Background

In this chapter, concepts that are relevant to this thesis are outlined and explained in order to help the reader understand the problem at hand.

2.1 Distributed Systems

A *distributed system* is a system that consists of multiple networked machines, often referred to as nodes, working together to resolve a particular problem [7]. This allows for a large number of resources to be used for performing more computationally demanding tasks, including solving complex equations, processing big data and training large scale machine learning models. Examples of distributed systems include cloud computing platforms (*Amazon Web Services*, *Microsoft Azure*), e-commerce systems (*Amazon*, *eBay*), search engines (*Google*, *Bing*) and social media platforms (*Facebook*, *LinkedIn*). Furthermore, a wide variety of architectural styles for distributed systems can be identified based on various perspectives, including but not limited to peer-to-peer, client-server, publish-subscribe and service-oriented.

The use of distributed systems in an organization's architecture can contribute to increased availability by preventing the existence of a single point of failure. Another advantage is scalability, since generally the number of machines can be adjusted to support the necessary workload. Other benefits to utilizing distributed systems can include efficiency, consistency and transparency [7]. As such, distributed systems can be encountered frequently in the architectures of mission-critical organizations.

2.2 Mission-Critical Systems

A *mission-critical system* is a system that is essential to the successful completion of an organization's intended objective [37]. If such a system fails and is no longer functional, there can be significant consequences to the organization, for example in the form of financial losses, decreased productivity, compromised security, or even physical harm. Examples of mission-critical systems can include traffic management systems in the transport domain, radar systems in the defense domain, payment systems in the electronic commerce domain, and many others. For systems that are also in the safety-critical domain, for example air traffic control or missile control systems, the consequences of downtime can be especially severe, including but not limited to property damage, environmental damage and harm to human life [40].

As a result of their essential role in an organization's operations, mission-critical systems require an imperceptibly short downtime period in the event of maintainance, hardware failure, network failure and other such causes of unavailability, in order to ensure that the larger operations that they support do not come to a halt.

2.3 Non-Functional Requirements

Mission-critical systems have stringent non-functional requirements for optimal and reliable operation. These requirements are outlined in this section in order to precisely define the problem addressed in this thesis.

Availability is a non-functional requirement that refers to the system's overall capacity to function, be accessible and have a quality performance with little to no disruptions or downtime [4]. In theory, the system can still be considered available even if its response to a request is slow, provided that a response is eventually ensured.

High availability (HA) is a stricter variant of availability where the system must be available 99.999% of the time, or five minutes and fifteen seconds of total downtime per year [44]. Mission-critical systems demand an especially high level of availability, with strict requirements on what is an acceptable duration of downtime in order to ensure the safe and secure operation of the system. Mechanisms for ensuring that the necessary levels of availability are reached for such systems are described in detail in Section 2.4.

In the context of distributed mission-critical systems, *consistency* refers to the ability of every component in the system to have the same correct, most up to date information on the state of the overall system, regardless of any failures that may occur. A simple example to illustrate the need for this requirement is bank transactions - if a transfer is made, all components in the system should reflect the changes, as opposed to responding with contradictory information when queried for the balance. Furthermore, in the event that there are inconsistencies, any subsequent transactions can also be affected.

Three types of consistency can then be identified [11]:

- *Strong* - near-immediate consistency is guaranteed with the aim of having the system function as if it is a single node processing tasks sequentially, usually by designating a particular node as the leader.
- *Eventual* - allows for a delay before all nodes have the same information, provided that no new updates are made to the state.
- *Strong eventual* - a stronger variant of eventual consistency, ensuring that each node that started in the same state will eventually converge to the same value, regardless of the order of updates.

Partition tolerance refers to a system's ability to continue to operate despite the presence of network partitions. Partitions can be considered scenarios of network failures and lost messages, while partition tolerance can be viewed as resilience in the event of such failure scenarios.

2.3.1 CAP Theorem

The CAP Theorem [29], first introduced by Eric Brewer in 2000, defines the relationship between the properties of consistency, availability and partition tolerance in a distributed

system or data store. The theorem establishes that there is a tradeoff between these properties, namely that at most two of them can be guaranteed in a distributed system.

For distributed systems, it is usually impossible to forfeit partition tolerance, as some form of network failure will inevitably occur throughout the life cycle of the system, meaning that the choice in practice is between consistency and availability. Furthermore, it is not necessarily a single decision for the entire system - a different tradeoff can be selected depending on the subsystem, the operation, the data involved, etc. [17]. For example, in a banking system, consistency of the data would likely be prioritized for transaction operations in order to avoid any mistakes in the user's balance, while for checking the account balance, availability might be preferred instead for a better user experience.

Given the crucial role of mission-critical systems within an organization, the decision to sacrifice one property in favor of the other can have significant consequences and should therefore be considered carefully.

2.3.2 Metrics

In [44], a comprehensive survey on the state of the art solutions for the challenge of ensuring availability in the cloud is presented by collecting different definitions of availability and downtime metrics, as well as defining a taxonomy with practices of different cloud providers. The taxonomy includes types of failures, availability mechanisms and metrics. The work of [44] is frequently used as reference for the contents of this chapter, including in this section.

According to [44], metrics for measuring the availability of a distributed system can be divided into three categories:

- Metrics related to failure:
  - *Mean Time to Failure (MTTF)* - the mean time it takes for the system to encounter failure.
  - *Mean Time to Repair (MTTR)* - the mean time it takes to repair a failed component.
  - *Mean Time between Failures (MTBF)* - the mean time between two failures occurring, equivalent to the sum of MTTF and MTTR.
- Metrics related to failure detection:
  - *Fault Detection Time* - the maximum time it takes to detect failure in the system.
- Metrics related to replication:
  - *Replica Number* - the number of duplicate components.
  - *Replica Creation Time* - the time it takes to duplicate the state of a component.
  - *Replication Frequency* - the time needed for state synchronization between the primary component and the duplicates.

As per the subject of the research questions, the metric most relevant to this project is fault detection time, referred to throughout this thesis as **failure detection time**.

There are two core metrics related to consistency that other, more complex metrics generally build upon [13]:

- *Staleness* - the degree to which a duplicated component is outdated compared to the most up to date component, measured in time or versions.
- *Ordering* - the order in which updates to the system are reflected on the duplicated components, measured by estimating the extent to which the established ordering is violated.

An example of a more complex consistency metric that utilizes staleness is the consistency-cost efficiency metric [18], which focuses on minimizing monetary costs in systems that use eventual consistency.

In the context of the CAP theorem, partition tolerance is generally considered in a binary perspective. The presence or absence of partitions determines which trade-offs are necessary between the three properties.

2.4 Availability Mechanisms

In this section, common mechanisms used to meet the requirements defined in Section 2.3 are outlined. In particular, the taxonomy defined in [44] is used as a reference for classification. An overview of the taxonomy can be seen in Figure 2.1.

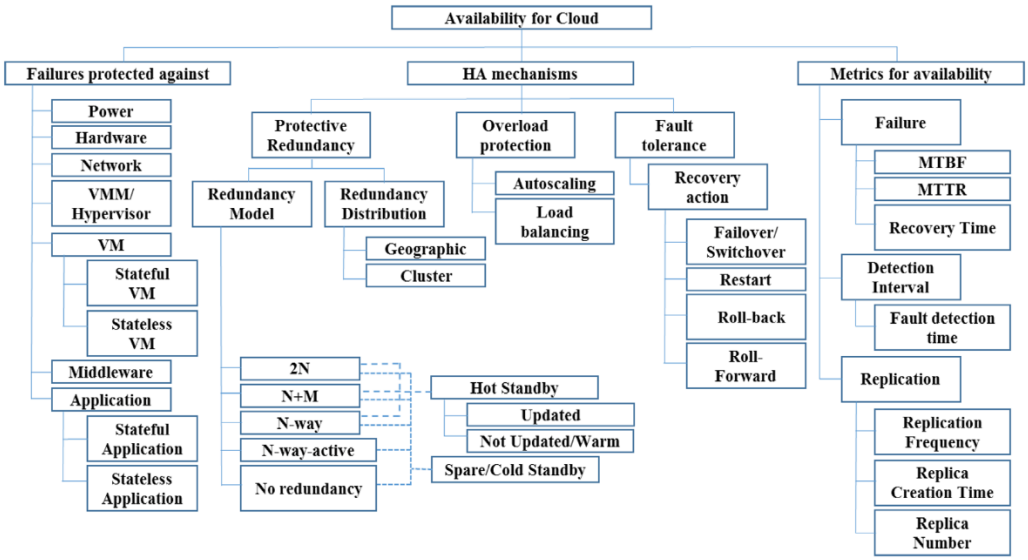

FIGURE 2.1: Nabi, M. et al. [44]. *A taxonomy for availability in cloud computing.*

2.4.1 Fault Tolerance

Fault tolerance is a type of availability mechanism that deals with the system's capability to continue functioning properly even when failure occurs [44]. By utilizing fault tolerance mechanisms, the system can detect failure as it occurs and handle it accordingly in order to ensure the continued operation and accessibility of the system with minimal downtime, thus contributing to improved availability. Two distinct phases of fault tolerance can be identified, namely failure detection and failure recovery. The respective processes associated with these phases are described in the following sections.

Health Monitoring

The failure detection phase of fault tolerance mainly consists of the *health monitoring* process. It refers to the set of mechanisms that are put in place with the aim of maintaining the system in a correctly functioning and well-performing state. This can include a monitoring process for detecting possible failures or anomalies, as well as a recovery mechanism for returning the system to its desired state by activating the specified recovery process [60].

During the monitoring process, a frequently used mechanism is that of *heartbeats*. Implementations can vary, but the general process involves sending heartbeat messages to the nodes in the system and waiting for a response [43]. If no response is received within a given timeframe, the node is assumed to have failed.

Failover

Failover is defined as a failure recovery process for providing uninterrupted availability in the event of failure of a system component by automatically switching to a backup component and redirecting traffic to it. After failure, the component is not used until it has been repaired, at which point it can again be made active, a process known as *failback* [56].

Fast failover is vital for the successful operation of mission-critical systems as it helps prevent the scenario in which a critical component is inaccessible [19]. Furthermore, it is the failure recovery strategy selected for the experimental stage of this thesis.

Recovery Strategies

Aside from failover, there are multiple options for recovery once a component fails, namely restart, rollback and roll-forward [44, 56]:

- *Restart* - the component is stopped and started again, usually in its initial state, in an attempt to eliminate the failure.
- *Rollback* - the system is reset back to a previous, functioning state, provided that snapshots of such a state have been collected prior to the failure occurring. Rollback has a generalized usage since it does not require any knowledge about the error, but is more resource-intensive due to the snapshot process.
- *Roll-forward* - if possible, the system is set to a new, functioning state, possibly after applying changes that could not be successfully executed beforehand. Roll-forward is generally more efficient, but requires application-specific knowledge.

Recovery strategies should be used with caution, as they do not necessarily resolve the original error, meaning that there is a risk of the same error causing a failure, even after the recovery strategies have been successfully applied [56].

2.4.2 Protective Redundancy

Protective redundancy refers to the mechanisms employed in the system to eliminate the possibility of a single point of failure occurring via the use of duplicated components. This can include redundancy models that handle the organization and rules related to the redundant elements, as well as redundancy distributions that define the geographical layout of the redundant components, forming availability zones (AZs) [44].

Standby Redundancy

Standby redundancy is an implementation of failover which makes use of duplicate components that are available to take over the tasks of the main component in the event of failure.

When using standby redundancy, each component can be considered to be in one of the following states:

- *Active* - the component is running, actively serving clients and handling traffic in the system.
- *Idle* - the component is running with some information about the current state of the system and is ready to take over without needing to be started first, but is not actively participating in the system.
- *Stopped* - the component is not running and would first need to be started if its usage is needed.

Three primary types of standby redundancy can be identified, namely cold, warm and hot:

- *Cold standby* refers to a standby redundancy setup in which the duplicated components are stopped until a failure occurs in the main component [42]. In such a standby, the duplicated component only begins operating after failure is detected and it is notified about the current state of the system.
- *Warm standby* is a type of standby redundancy in which idle duplicates of the critical process are running and ready to take over in case the active component experiences a failure. Using warm standby, some short amount of downtime is expected, but can be optimized. The switch from idle to active state can happen relatively quickly, as the idle component is essentially already running and available and only needs to be notified that it needs to take over as the active component [47]. This means that optimization needs to be aimed at reducing the time it takes for the failure to be detected and communicated to the relevant duplicate component.
- *Hot standby* also has duplicate components running, but these instances are not as passive as those implemented in warm standby. Instead, the duplicates are in constant communication with the primary component and are actively serving clients. As a consequence, the duplicates are ready to take over immediately if failure occurs. However, by being active they are more likely to encounter failure faster due to already being in use [42].

Using cold standby, the duplicate components are less likely to suffer from failure before becoming active due to prior usage, but the result can be a slower restoration of the system to a properly functioning state. Hot standby, on the other hand, offers a much faster recovery time at the expense of reliability [42]. Furthermore, the recovery process occurs much faster when using hot standby compared to warm standby.

2.4.3 Overload Protection

Overload protection is a system's ability to handle incoming traffic and distribute resources in such a way that performance is not severely degraded or disrupted [44]. Common mechanisms for achieving overload protection are load balancing and autoscaling.

Load Balancing

Load balancing refers to the process of distributing incoming traffic and workload to the suitable system component that should handle it [6]. This process is used to improve scalability and resource utilization by ensuring that no component is overloaded or idle. Load balancing enables HA by allowing the system to function correctly and efficiently without being overwhelmed by traffic.

Autoscaling

Autoscaling is a mechanism often used in the cloud to automatically adjust the available computational resources, depending on the load that the resource needs to handle at any given time [50]. Autoscaling builds on the idea of load balancing and ensures that the system can optimally handle varying amounts of incoming traffic without the need for manual intervention.

2.5 Kubernetes

In this section, background information on Kubernetes relevant to this thesis is provided, in particular its components, architecture and availability mechanisms.

2.5.1 Architecture

Kubernetes [8], often abbreviated as K8s, is an open-source platform, originally developed by Google, that is used for container management. In other to support this task, Kubernetes' features include service discovery, storage orchestration, self-healing, automatic scaling based on demand, configuration management, load balancing, and many others.

A diagram of a typical K8s-based architecture can be seen in Figure 2.2. The result of a setting up Kubernetes is a *cluster*, which consists of at least one node. Each node is a set of worker machines (physical or virtual) responsible for running containers, while a control plane is used to manage the cluster, including responding to events within it, facilitating internal communication, workload allocation, etc.

Containers are wrapped around elements called *Pods* that are hosted by the nodes. Each worker node is managed by an element called a *kubelet*. In order to manage the networked communication with a node, a *kube-api* component that keeps track of networking rules is running on each node. Worker nodes also manage additional Kubernetes resources such as Deployments, DaemonSets, etc.

The frontend of the control plane is the *kube-apiserver* component, which exposes the Kubernetes API and manages the interactions between other components in the cluster. The kube-apiserver processes and validates any requests made through the Kubernetes API.

Data about the cluster, including its state and configuration, is stored via the open-source *etcd* [26] key-value store, which provides a consistent and highly available way to store data for distributed systems. Furthermore, etcd can be used for service discovery, container and cluster coordination, workload distribution, etc.

The scheduling, i.e. allocation of resources to Pods, in Kubernetes is performed by the *kube-scheduler* component. Various controllers, such as node controllers, job controllers, etc., are ran within the *kube-controller-manager* component.

Another important aspect of Kubernetes's functionality is that of *services*. Services are used in order to map network traffic to Pods, from both within and outside the cluster. This is due to the fact that in a Kubernetes cluster, an application is not expected to be available at the same IP address constantly - Pods frequently get replaced or rescheduled, and furthermore different replicas of the same application are usually available, making a static IP address difficult to work with. Services are designed to address this problem by routing incoming traffic to a suitable Pod, without requiring any extra knowledge other than the service's name.

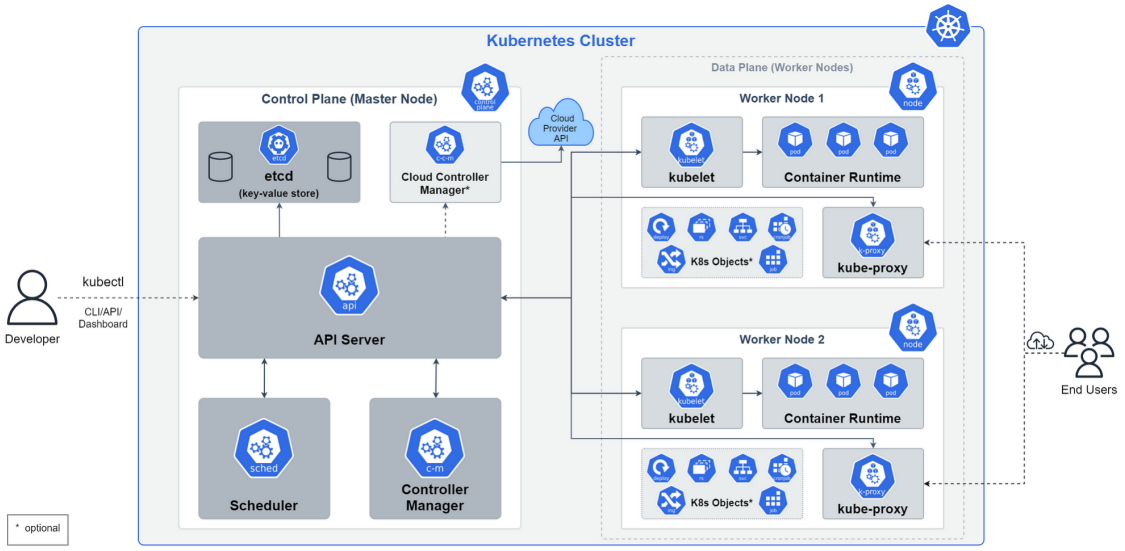

FIGURE 2.2: Patel, A. [48]. *Kubernetes Architecture and Components.*

2.5.2 Availability Mechanisms in Kubernetes

Kubernetes provides self-healing and redundancy as built-in features for ensuring recovery in failure scenarios, as detailed in the Kubernetes documentation [8].

Self-Healing

Self-healing refers to the ability of a system to recover on its own whenever failure is detected. In Kubernetes, this is done by continuously monitoring the state of the cluster components, killing unresponsive containers and restarting any that have failed.

The monitoring aspect is achieved by periodically performing two types of probes, namely liveness and readiness. Liveness probes are used to check for container responsiveness in order to detect issues such as deadlocks, crashed containers, zombie processes, etc. Readiness probes are used to determine whether a container is ready to accept traffic. If it is not, traffic is temporarily not sent its way for as long as the container does not respond to the probe.

Redundancy

Redundancy is a crucial aspect for achieving HA [4], enabling protection against a single point of failure in the system by duplicating critical elements, as discussed in Section 2.3.

Several mechanisms can be employed in Kubernetes, often in combination, in order to achieve redundancy. This includes ReplicaSets (generally managed by Deployments), load balancing, horizontal pod autoscaling, redundancies at the master node level, topology spread constraints, etc.

A *ReplicaSet* is a type of Kubernetes resource that is used to ensure that there is a specified number of instances of a particular Pod running at all times. In the event that any one of these replicas fails, it is the ReplicaSet's responsibility to ensure that another Pod of the same type takes its place, thus ensuring that the required number of instances is running. A Deployment is a type of resource that can be used to manage a ReplicaSet, along with other higher-level features, and, as such, it is recommended to use Deployments over ReplicaSets [8].

Deployments and *DaemonSets* are two important redundancy resources for the architecture of this project - the former is used to flexibly manage the number of replicas of a given application, while the latter can be used to ensure that a component is available on every node in the cluster.

A *HorizontalPodAutoscaler* is a type of resource that automatically adjusts a scalable resource based on the demand by deploying more Pods. Vertical scaling, on the other hand, involves allocating more resources, such as CPU and memory, to the already running Pods [8]. Horizontal pod autoscaling ensures that the workload resource is always flexible enough to meet the demand required by the system.

Topology spread constraints allow for the custom definition of the way Pods are spread across the cluster. These constraints can be regarding nodes, AZs, and other topologies. This can contribute towards achieving redundancy and HA by ensuring that different Pods are spread across different physical locations, which can protect against single point failure. Additionally, the use of topology spread constraints can allow for more efficient use of resources [8].

Failure Detection Performance

As mentioned in Section 1, the failure detection capabilities of Kubernetes are not always sufficiently fast and thus cannot meet the HA requirements of mission-critical systems.

The research done by [3] explores the capabilities of Kubernetes with its built-in mechanisms for managing the availability of cloud applications. The paper measures metrics such as reaction, repair, recovery and outage times for different failure types, as well as the effect that using redundancy mechanisms has on availability. The paper finds that Kubernetes with its default settings can have significantly high outage times, particularly in the event of node failure. Furthermore, the paper includes a comparison between the availability capabilities of Kubernetes with those of other component managers, such as OpenSAF, and found the results to be comparable. Part of the experimental phase of this thesis involves measuring the failure detection time of Kubernetes, further confirming the findings of [3]. The results of these experiments can be seen in Section 4.4.

The work of [61] builds onto the research of [3], establishing a method called Fast Fault Detection Manager (FFDM) for improved fault detection and recovery of mission-critical systems on the cloud. It does so by taking into consideration both the application and node level, as well as by using parameters in Kubernetes other than only the default ones. The results that the FFDM method is an improvement over the previously defined method, performing more than three times better at fault detection. However, this method is not

fully applicable to the case described in this thesis, as its main strategy involves failure detection at the VM-level.

Another work that addresses the availability capabilities of Kubernetes is that of [12]. The work presents a comprehensive analysis and classification of failures in Kubernetes, as well as a framework dedicated to recreating real-world failure scenarios with the aim of detecting potential failures in the system before they occur. The paper's findings also indicate that individual errors in the system can propagate throughout the cluster, causing it to fail, even if resiliency mechanisms are implemented. Furthermore, misconfigurations and label errors when tracking dependencies between objects in the system cause a large percentage of the observed failures by overloading the system.

2.6 Data Distribution Service (DDS)

The *Data Distribution Service (DDS)*, standardized by the Object Management Group (OMG), is a data-centric middleware protocol and API standard that is especially well-suited for real-time systems [45]. The protocol is designed to address issues such as scalability, efficient resource utilization, flexibility and security. In particular, it is widely used by mission-critical systems due to its ability to provide efficient low-latency communication in a stateless manner [46].

DDS accomplishes this by utilizing a publish-subscribe model of communication, allowing applications in the system to produce and update data, and publish it to a shared space, while entities that are interested in a particular topic can subscribe to it in order to receive the relevant data, as opposed to requiring a direct communication between the different participants in the system. The model used by DDS ensures an efficient use of bandwidth and processing power [45]. Furthermore, DDS enables a decoupled form of communication between different components, thus lowering the complexity of the overall system, while its use of topics for the publish-subscribe model eliminates the dependency on Kubernetes Services for IP-based communication [39].

These characteristics make DDS a suitable protocol for communication within a mission-critical distributed system, as it is designed to meet the necessary requirements for performance and reliability [39]. As such, its usage must be considered when selecting a fitting solution.

2.6.1 Cyclone DDS

Multiple implementations of DDS exist [53], including Eclipse Cyclone DDS, eProsima Fast DDS, OpenSplice DDS, etc. For this research, Cyclone DDS [55] was selected as the preferred option, in particular its Python binding. The factors considered when making this design choice mainly included ease of use and relevance to mission-critical systems. For the latter factor, OpenSplice DDS can be a suitable choice, as it is also preferred by Thales, however, it was deemed too complex for a system that is as small in scale as this project. Cyclone DDS, on the other hand, offers a fairly simple to use Python binding without sacrificing any necessary complex features, and was thus considered a suitable fit for this project.

2.7 Gossip Protocol

A *gossip protocol* is a type of communication protocol that propagates data throughout a cluster of networked nodes, by way of each node periodically exchanging information with some of its peers [41].

The *fanout* of a gossip protocol indicates the number of rounds for which each node spreads a piece of information, after which the node becomes dormant with respect to that particular data [16]. Typically, the fanout is set to $\log(N)$, where N represents the number of nodes in the system. The *mixing time* of a gossip protocol refers to the time it takes for each node to receive the propagated information, typically within $\mathcal{O}(\log(N))$.

More specifically, the following characteristics must be present in a protocol for it to be considered a gossip protocol [16]:

1. The protocol fundamentally consists of periodic interactions between pairs of processes.
2. Interactions are of limited, small size.
3. Interactions involve a change of state in one or both nodes, influenced by the state of the other.
4. Reliability is not a guarantee.
5. Interactions are infrequent and therefore inexpensive.
6. Peer selection is random.

Furthermore, three types of gossip protocols can be identified [16]:

- *Dissemination* - used for spreading information in a cluster.
  - *Event dissemination* - perform multicasts and report on events periodically.
  - *Background data dissemination* - data about the nodes is continuously propagated throughout the cluster.
- *Anti-entropy* - used for repairing duplicated data by comparing the duplicated components.
- *Aggregate computation* - protocols that examine the nodes and derive from the collected data a conclusion about the system as a whole.

Dissemination gossip protocols are the most relevant, due to the fact that the outlined problem relates to the spread of information regarding the state of the nodes in the cluster.

Chapter 3

Potential Solutions

As established in Chapter 1, the use of Kubernetes alone for failure detection and communication between nodes is not always sufficient to fulfill the strict availability requirements of mission-critical systems.

In this section, RQ1 and its subquestions are answered by providing a background on potential solutions to the problem that were considered for a more efficient failover in the event of failure in a distributed system. Furthermore, these technologies are compared and one is selected to proceed with the experimental phase of this thesis.

3.1 Approach

The approach to answer the first research question mainly involved a literature review with three phases:

1. An exploration of literature on failure detection and recovery, in order to accurately define the challenges in the domain, as well as their relevant concepts.
2. A literature search to identify technologies that could act as suitable solutions to the problem, supported by stakeholder interviews at a company in the relevant domain.
3. A study of these technologies in more detail with the aim of establishing their advantages and disadvantages, as well as their compatibility with Kubernetes.

The first phase required a more systematic approach, while the second phase was based mainly on stakeholder interviews and information from the papers found during the first phase. The third phase focused on finding specific information about the technologies identified in the second step, where the queries used were quite straightforward and self-explanatory.

In the first phase, the repository that was mainly used to query for relevant research is Google Scholar [30], as it is a source that provides extensive and reliable peer-reviewed results. In order to find relevant papers, associated concepts were searched for via the following keywords that encompass them:

- *Health monitoring*, *failure detection*, *fault detection*, *failover*.
- *Distributed*, *cloud*.
- *Availability*.

- *Mission-critical.*
- *Kubernetes.*

The primary string used was:
`("health monitoring" OR "failure detection" OR "fault detection" OR "failover") AND ("distributed" OR "cloud") AND "availability" AND ("mission-critical" OR "mission critical") AND "kubernetes"`.

Since the set of retrieved literature was quite large (387 results), not everything could be closely examined for this research. Therefore, the results were narrowed down to the most relevant. In particular, papers taken into account included research with a higher number of citations, indicating importance in the field, as well as recent works that are likely to be more up to date, while having less time to be cited.

Furthermore, the titles and abstracts of the publications were examined to eliminate irrelevant results. The following inclusion and exclusion criteria were considered:

- Inclusion criteria:
  - Focuses on distributed systems.
  - Primarily discusses fault tolerance or a fault tolerance phase (failure detection or failure recovery).
  - Addresses the non-functional requirement of availability.
  - Relates to Kubernetes in some way.
  - Applies to mission-critical systems.
- Exclusion criteria:
  - If fault tolerance is only a secondary aspect, the publication is discarded.
  - Publications with emphasis on other technologies irrelevant to the problem, such as AI, 5G, etc., were discarded.
  - Publications in languages other than English were discarded.

The remaining papers were read in detail and their findings were used in this thesis. Via this approach, potential solutions were identified, answering RQ1a. The candidates are namely Serf, Consul and ZooKeeper, discussed in more detail in the following sections.

3.2 Serf

Serf [35], developed by HashiCorp, is a decentralized, lightweight protocol designed for communication between nodes in a distributed system, in particular for the purposes of cluster management, failure detection and orchestration. Nodes communicate via an efficient gossip protocol, meaning that updates to the node membership are relayed to any node in the system, after which the information is propagated through the cluster until each node has the latest membership information.

For membership management and failure detection, Serf makes use of the memberlist library [33], which itself is based on the SWIM protocol [23]. Serf allows for fast, scalable failure detection via a random probing technique, as each node can check on the status of its neighbours periodically and notify them if failure is detected. Once a node has been

detected as having failed, Serf will continue to periodically attempt to reconnect to it in case the node in question has recovered.

Serf offers a set of powerful features, including custom event and query propagation, allowing configuration updates, deployment triggers, health checks, etc. to spread within seconds throughout the cluster. Queries expect a response, while events are fire-and-forget. In the event of network issues or partitions, a best-effort attempt is made to deliver these messages.

Additionally, custom event handlers are a feature that can be triggered by events within the cluster such as whenever failure occurs, allowing for a wide range of customizable actions. The use cases of event handlers and queries are plentiful - load balancer management, triggering deployments, observing the health of the cluster, building a service discovery mechanism, etc. The option to write event handlers in Python is also available. Each of the aforementioned features can be enhanced with Serf's ability to mark its agents with custom tags, further adding flexibility to the cluster management.

On its own, Serf manages only membership, failure detection and custom events, making it an especially lightweight option for the outlined problem. Furthermore, Serf only takes up 5-10MB of resident memory and mainly communicates over UDP with a limited message size. Another relevant aspect of Serf is that it ensures strong eventual consistency within the cluster.

3.3 Consul

Consul [32], also developed by HashiCorp, is an open source tool used for secure, automated networking and service discovery in a distributed architecture. In particular for Kubernetes, Consul can act as a service mesh for monitoring and managing the networked communication between different services. It enables service discovery, health monitoring and distributed configuration management in a scalable, secure and resilient way. Consul further offers features such as support for multiple clusters (also referred to as datacenters) and an API gateway for defining traffic and authorization policies.

Consul has a wide variety of use cases, quite a few of which overlap with Serf. Its uses include service discovery, load balancing, automation of networking tasks, authentication and encryption, health monitoring, metrics collection, and many others. Companies that make use of Consul in their operations include *eBay*, *Capital One* and *SAP* [49].

Consul uses a centralized architecture with a central control plane component that maintains a registry keeping track of services and their IP addresses, while different agents act as either client or server, with a recommended 3-5 servers per cluster to ensure higher availability, though this comes at the expense of a slower performance due to the use of a consensus algorithm. Via the consensus algorithm, the cluster elects a single server agent to be the leader, responsible for processing any transactions and queries. Clients, on the other hand, are responsible for the health monitoring process by reporting status updates to the cluster.

Consul is built on top of the Serf library and uses the same gossip protocol for failure detection. However, unlike Serf, it provides strong consistency, uses a consensus protocol has a centralized architecture.

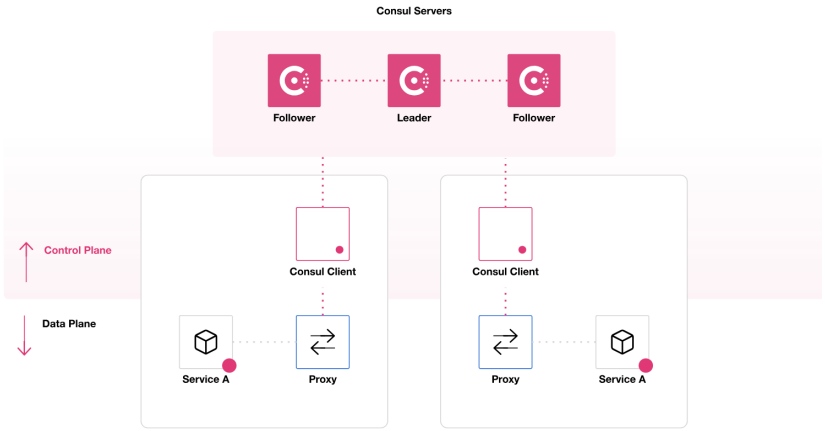

FIGURE 3.1: Architecture and components of Consul [32].

3.4 ZooKeeper

ZooKeeper [27] is a centralized service maintained by the Apache Project Foundation, dedicated to coordinating processes in a distributed system via a shared namespace of data registers. Its features include keeping track of status, group membership and configuration data, naming, as well as synchronization among processes. ZooKeeper or forked variations thereof is used by companies such as *Facebook*, *Yahoo!* and *X* (formerly known as *Twitter*).

ZooKeeper aims to assist in the building of distributed applications by eliminating the need for each such system to create a coordination service from scratch, and thus avoiding otherwise prominent issues such as deadlock and race conditions. It further offers support for features such as failure detection, leader election and distributed locks.

ZooKeeper's architecture is based on a shared hierarchical namespace consisting of data registers, referred to as znodes. These registers contain information regarding status, configuration, location, etc. The structure of ZooKeeper resembles that of file systems, but with data kept in-memory, ensuring a high throughput and low latency. Furthermore, ZooKeeper provides its users with a set of primitives, such as locks, queues, etc., that can be used to implement more complex services.

ZooKeeper is a reliable, high performance service for large distributed systems, offering robust management, synchronization, and configuration capabilities. However, despite these features, it is better suited for coordination rather than service discovery, given its prioritization of consistency over availability. Furthermore, it can be complex and difficult to maintain compared to other solutions [54]. ZooKeeper provides strong consistency and uses a heartbeat mechanism, as described in Section 2.4.1, for checking liveness [35].

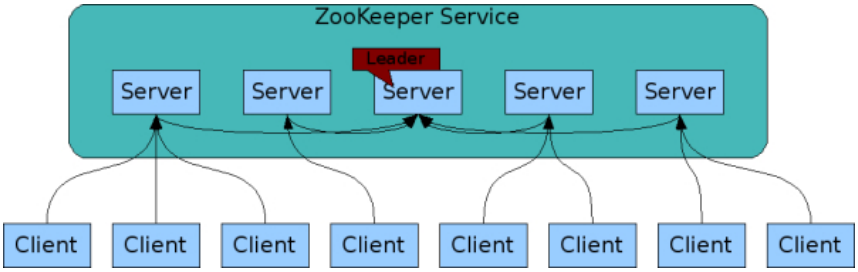

FIGURE 3.2: Architecture and components of ZooKeeper [28].

3.5 Comparison

In this section, the proposed technologies are compared through the advantages and disadvantages that they can offer for monitoring the health of a highly available mission-critical system, thus providing an answer to RQ1b.

The findings of [27] and [51] are adjacent to the work done in this section, as they deal with comparisons between the technologies discussed in this thesis.

In [31], a comparison is made between multiple methods for the purpose of failover in a mission-critical system on the cloud, specifically for the purpose of migrating a system running in a private data center to the public cloud, where different failover mechanisms are required. In particular, two technologies, namely ZooKeeper and etcd locks, are compared experimentally with two scenarios. In the first scenario, the primary component is able to notify the duplicated component about the failure before termination, while in the second the failure occurs without prior notification. The work finds that the ZooKeeper solution results in a faster failover time compared to the etcd solution. For this reason, etcd was not considered in the comparison done for this thesis.

The work of [51] presents an overview of the state of the art of the software stack of distributed computing. The paper includes a comparison between the three technologies compared in this thesis as well, alongside other software for distributed computing. This work provides a thorough classification based on abstraction layers of the latest software available at the time of its writing, resulting in the comparison and taxonomy of more than 150 technologies. The comparison done in [51] differs from that of this thesis, as it deals with a broader context compared to that of the problem specified here and thus does not make a complete comparison of all aspects that are relevant in the context of mission-critical distributed systems orchestrated via Kubernetes.

For the comparison of this thesis, an overview of the features of each approach can be seen in Table 3.1. Each aspect in the table is discussed in the following sections. Scalability and ease of use are not included in the table and are only discussed within the text due to a lack of a brief way to summarize them.

3.5.1 Quorum

A *quorum* [58] is a mechanism used to ensure consistency within a distributed system with potential network partitions, in particular when redundancy is in use. It does so by requiring that a specific number of nodes reach a consensus on a decision that affects the state of the system before any action can be taken. The *resilience* of a quorum system refers to the maximum number of nodes that can fail, while still being able to achieve a

	Serf	Consul	ZooKeeper
No quorum required	✓	✗	✗
Decentralized architecture	✓	✗	✗
Type of consistency	Strong eventual	Strong	Strong
Failure detection protocol	Gossip	Gossip	Zab
Kubernetes compatibility	Unknown	✓	✓

TABLE 3.1: Comparison of Serf, Consul and ZooKeeper.

quorum. A distributed system cannot have a resilience greater than $\lfloor \frac{n-1}{2} \rfloor$, i.e. no more than the specified number of nodes can fail.

Although quorums are frequently employed in the coordination of distributed systems, their use can present a problem for some mission-critical systems. In particular, the deployment of some such systems can be restricted to a relatively small number of availability zones, potentially as few as two, due to topological constraints. Examples of such systems include those in the naval defense domain, the aerospace industry, the railway industry, etc.

Systems with these geographical limitations introduce some complications. In any setup, a quorum cannot be reached with an even number of nodes, as it is impossible to reach a majority vote in this way. The alternative then is to utilize an odd number of nodes. However, in the scenario that nodes are divided between two availability zones, as can be the case for the mission-critical systems described previously, one zone would have an even number of nodes and the other - odd. If the entire availability zone with an odd number of nodes suffers a failure, then the remaining functional nodes are even, still making it impossible to reach a quorum.

For systems that are set up in such a way, the use of a technology requiring a quorum can be undesirable, since it introduces problems regarding availability. This needs to be taken into consideration when selecting a suitable technology to resolve the presented problem. Out of the coordination technologies that are examined in this thesis, only Serf does not require a quorum in order to act. ZooKeeper uses the Zab consensus algorithm to implement quorums, while Consul uses the Raft consensus algorithm.

3.5.2 Centralization & Consistency

Other relevant aspects when considering which technology is best suited to address the problem defined in this paper are those of centralization and consistency type.

Centralized technologies, such as Consul and ZooKeeper, rely on a particular node that is elected as a *leader* to coordinate the remaining nodes, referred to as *followers*, in the cluster. This can introduce several availability-related issues. In particular, the leader node represents a single point of failure - if it fails, the coordination of the rest of the nodes will experience an interruption as well. Assigning multiple nodes as leaders can mitigate this problem, but introduces a potential scalability issue [25].

Furthermore, the type of consistency that the technology prioritizes also has a significant impact on its suitability as a solution to the problem. According to the CAP theorem, as defined in Chapter 2, in partitioned systems there is an inherent tradeoff between availability and consistency.

Technologies such as Consul and ZooKeeper offer strong consistency by requiring a quorum

prior to committing an action. This means that in the event of failure where a quorum cannot be reached, they would prioritize consistency across all nodes in the system over availability. This makes Consul and ZooKeeper less suitable for systems that have HA requirements, as they sacrifice availability. Serf, on the other hand only guarantees strong eventual consistency, thus ensuring that availability is prioritized instead of consistency.

3.5.3 Scalability & Ease of Use

An aspect that should be considered when selecting a tool for monitoring distributed systems is the degree to which its performance is negatively affected when the system increases in size.

For Consul, increasing the number of deployed services can cause a slowdown in performance. As such, HashiCorp's recommendation [34] is to limit the number of clients per datacenter to a maximum of 5000, both to optimize performance and to limit the impact of server failure by reducing the number of affected clients. However, the advice is not one-size-fits-all and depending on how dynamic the node activity is, the recommended number of nodes per datacenter can increase or decrease.

When using ZooKeeper, increasing the number of servers proportionally increases the system's capacity to handle read requests, since these requests are processed locally at each server [38]. Write requests, on the other hand, inherently do not scale as well due to the centralized nature of ZooKeeper [21].

On the other hand, Serf's decentralized architecture and reliance on the gossip protocol makes it an efficient solution even when scaling up to a higher number of nodes is required. The usage of the gossip protocol ensures a low message load per each node that is not affected by an increase in the group size [54].

In terms of ease of use, ZooKeeper clusters are considered quite difficult to deploy and manage, often resulting in issues with misconfigurations [54]. The process of making changes to the deployment can also be error-prone, inefficient and requires manual effort to achieve [21]. Consul, on the other hand, can be viewed as an extension of Serf, in particular with regard to the gossip protocol, offering additional features that Serf does not, such as service discovery, key/value store, health checking, etc. [35]. However, for the purposes of the problem defined in this paper, these additional high-level features are not needed and Serf's features can be sufficient as a solution. As such, Serf can be considered an alternative to Consul that is simpler to use.

3.5.4 Kubernetes Compatibility

In this section, the compatibility of each of the three technologies with Kubernetes is discussed, in order to answer RQ1c.

Consul can be run directly on Kubernetes, as well as any other tool built for Kubernetes, and it supports any Kubernetes runtime. Furthermore, Consul's built-in integrations with Kubernetes include the Helm chart, the Consul K8s CLI, the Consul Service Mesh and service syncing [32].

ZooKeeper can be run on Kubernetes via the use of the StatefulSets, PodDisruptionBudgets, and PodAntiAffinity resources, with a detailed tutorial on how to accomplish this being available on in the Kubernetes documentation [8].

There is limited information in literature regarding Serf's compatibility with Kubernetes. A further exploration of the problem is needed outside of this thesis.

3.5.5 Selected Solution

Taking each of the above-discussed aspects into consideration, the technology selected to proceed with an experimental comparison is **Serf**, as its decentralized nature, prioritization of availability and lack of quorum requirements make it a suitable fit for mission-critical systems with strict HA expectations, thus answering RQ1.

Chapter 4

Experiments

In this section, the approach taken to answer RQ2 is outlined. In particular, the in-depth workings of Serf's failure detection algorithm are described, as well as the architecture of the solution. Furthermore, the results of the experiments are presented and their implications are discussed in order to make a comparison between the use of Kubernetes alone and Kubernetes in combination with Serf for the purposes of failure detection. In general, the emphasis of this research is on especially small systems with 2 nodes, hence the focus on the algorithm's workings in such a scenario. That being said, the use of a larger number of nodes is also explored, albeit less extensively.

4.1 Related Work

To start with, two works are worth pointing out for their adjacency to the experiments performed for this thesis.

The work of [54] explores the service discovery problem for distributed systems and proposes the use of a decentralized custom solution called Serfnode, based on the Serf technology. Serfnode works by wrapping Docker containers and then performing service discovery, monitoring and self-healing by communicating with other Serfnode containers via a gossip protocol. The paper presents a solution to the file synchronization problem using Serfnode, demonstrating the extensibility of the approach. According to the authors, Serfnode is expected to scale well, but is nevertheless best suited for smaller networks. Unlike this thesis, the work does not focus on failure detection and fast failover, but rather on the service discovery problem.

Additionally, of note is the work of [52], which, similarly to this thesis, explores the problem of high availability orchestration with Kubernetes in mission-critical systems, specifically for on-premise systems in the defense domain. The paper's contribution includes five architectural designs dedicated to resolving the problem, using either Kubernetes Federation or single cluster designs. Each design is evaluated by domain experts via a questionnaire, with its findings indicating that federated architectures are too complex for the problem and thus single cluster options are preferred. Two single cluster designs are prototyped and tested, showing promising results. The work differs from this thesis in its focus on designing a suitable cluster architecture to resolve the problem of high availability, rather than on optimizing the technologies used to communicate failure.

4.2 Serf

To start with, a more detailed exploration of the relevant parts of Serf's algorithm is presented in order to answer RQ2a.

4.2.1 Memberlist Library

In order to be able to understand how the results of the experiment were obtained, the workings of Serf's failure detection mechanism are explored. More specifically, Serf utilizes the memberlist library [33] for failure detection. Its primary function is to manage cluster membership, including failure detection, by using a gossip protocol.

An important work that closely accompanies this research is that of [23], in which the SWIM protocol is presented for cluster membership management. This protocol is the basis for failure detection in the memberlist library. The SWIM protocol notably separates the membership updates and the failure detection components, unlike traditional algorithms that use heartbeats. Another important achievement of the protocol is that its use of the gossip protocol ensures that scaling cluster size does not affect the expected failure detection time, nor the message load of any member. The key workings of SWIM are de facto detailed in this section via the explanations for memberlist.

The memberlist library keeps a record of all members in its cluster and performs its failure detection functionalities on an individual member basis. The library allows for the tuning of its parameters, including those that affect the performance of the failure detection process. Furthermore, three key sequential stages of the failure detection process can be identified - selection, probing and suspicion. The functions within the memberlist code responsible for these stages are *probe*, *probeNode* and *suspectNode*, respectively.

4.2.2 Parameters

After examining the algorithm, the following parameters of memberlist were found to have an impact on failure detection:

- *Suspicion multiplier (SuspicionMult)* - indirectly determines the duration of the suspicion stage, i.e. the time for which a potentially failed node is marked as suspect before being considered dead. In practice, this duration is determined by the suspicion timeout variable, which is calculated by the following formula, where N is defined as the number of nodes in the cluster:

  $$SuspicionTimeout = SuspicionMult \times \log(N+1) \times ProbeInterval$$

  The purpose of thesuspicion multiplier variable is to allow the suspicion timeout to scale properly by anticipating a longer propagation delay with a larger cluster size. By default, this value is set to 4. However, for this research, the aim is to minimize the failure detection time, and so it was set to 0 for the optimized version of Serf, thus essentially eliminating the suspicion stage.

- *Probe interval* - the duration in between random probes. Not as immediately obvious, however, is the fact that this parameter also controls the duration of a probe once it has started, especially in the scenario in which there are only 2 nodes. This is discussed in more detail in Section 4.2.4. Reducing this parameter can reduce the failure detection time at the expense of bandwidth, but since the focus of this

project is on smaller systems, this is not much of a concern. The default value of this parameter is 1 second, reduced to 100 milliseconds for the optimized version.

- *Probe timeout* - determines the amount of time to wait for a response from a probed node before moving on to the next steps of the probing stage, again described in Section 4.2.4. By default, this parameter is set to 500 milliseconds, and was reduced to 50 milliseconds in the optimization. Additionally, it is a requirement by memberlist is to set this parameter to a value in the 99th percentile of the round-trip time of the network.
- *Indirect checks* - defines the number of nodes that will be asked to perform an indirect probe of the targeted node, after the direct probe by the self has not received an acknowledgment. This aspect is only relevant for scenarios in which more than 2 nodes are in use, as it does not include the self, nor the node being probed. For the main experiments with 2 nodes, it was left as its default value, which is 3, while for experiments with multiple nodes, it has been adjusted without a fixed value. The role of this parameter is discussed in more detail in Section 4.4.
- *Gossip interval* - the time between gossip messages being propagated throughout the cluster. This only occurs for messages that have not been piggybacked via probes. Considering that this implementation prioritizes fast failure detection time, and thus probes occur as frequently as possible, the need for gossip is minimal and thus its impact on the system is also negligible. However, it is still worth noting that this parameter would normally be relevant when adjusting the parameters that affect failure detection time in Serf. Its default value is 200 milliseconds.

The optimized values were chosen with the intent of making the process as fast as possible, considering the research questions that this project aims to address. This implies that such a configuration could potentially present problems with regard to consistency, including scenarios in which a false positive occurs. That being said, no such incident was encountered during the experiments. Furthermore, setting the parameters at lower values than those listed above was not permitted during the build process, as it resulted in failed tests. This could imply that false positives are unlikely to occur given these values, while lowering them past what is permissible by memberlist could be problematic.

4.2.3 Selection Stage

The first stage of memberlist's failure detection component is the selection process, which determines what node in the cluster other than the self is to be probed in the next stage. The selection stage is entered approximately every *probe interval* units of time and in each run of the respective function, at most one node is probed. An activity diagram to support the reader's understanding of the code can be seen in Figure 4.1.

Two variables are of importance in this function - *numCheck*, which keeps track of how many nodes have been considered as candidates for probing, and *probeIndex*, which keeps track of the current candidate within the membership list. Provided that neither of these variables is larger than the number of nodes, as denoted by N, the function proceeds to check whether the candidate node should be skipped, in the case where it is the same as the node performing the probe, or in the case where the candidate is already dead. If the node is not to be skipped, it is passed on to the probing stage.

The exact reasoning behind the way in which the *numCheck* and *probeIndex* variables are modified inside the function is unclear, and could not be intuitively guessed. An important

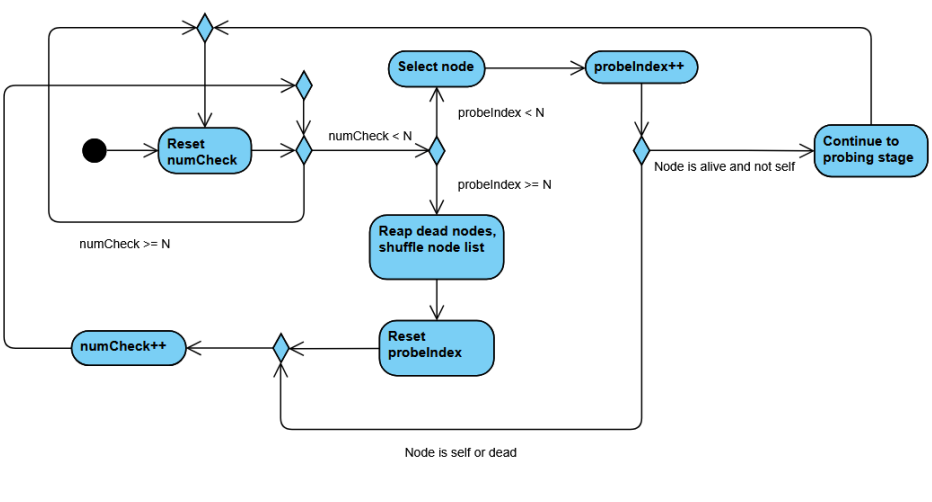

FIGURE 4.1: Activity diagram depicting the process of the selection stage in memberlist.

observation, however, is that it can create a delay of at most around one *probe interval* unit of time, since the modification of the variables occasionally ensures that one probing round is wasted without actually probing a node. This is important to consider when estimating the maximum amount of time in which failure can be expected to be detected, as discussed in Section 4.4.

4.2.4 Probing Stage

The probing mechanism for failure detection of Serf can be seen on the sequence diagram in Figure 4.2. The left-hand side shows the behaviour when the system is operating as expected, while the right-hand side demonstrates the scenario in which a failure occurs at Node B, which is subsequently detected by Node A. This depiction is intended for scenarios with exactly 2 nodes, as the process has an extra step of indirect probing when more nodes are involved. This process is also described in extensive detail in [23], on the basis of which both Serf and memberlist are implemented.

The probing process involves the following steps (some details have been omitted, as they have no relevance to the failure detection time):

1. In the event that the health of the cluster is known to have deteriorated, the probe interval is scaled. Based on observation, no change to the probe interval is applied before any failure has occurred, so for the experiments performed in this research, this aspect is generally irrelevant.
2. A timer with the duration of *probe interval* is started. Another time with the duration of *probe timeout* is also started.
3. Node A attempts to ping Node B directly.
4. If an acknowledgment is received, the probing stage ends and after *probe interval* units of time, the selection stage starts anew.
5. Once the probe timeout timer runs out without a response from Node B, the process of indirect pings is started. At this point, there are two possibilities:

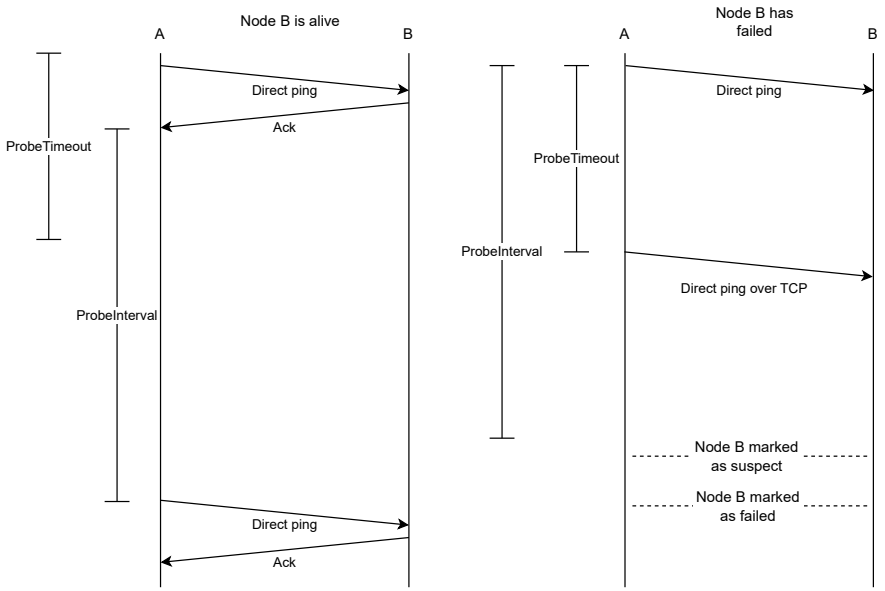

FIGURE 4.2: Sequence diagram depicting the probing mechanism for failure detection of Serf.

(a) In the case of 2 nodes only, this step is essentially skipped, as there is no other node to request an indirect probe from.

(b) In the case of more than 2 nodes, an *indirect checks* number of nodes are randomly selected from the membership list and each of them sends a ping to Node B. If a response is received at that point, the probing process ends.

6. An additional check is made by sending a ping over TCP instead of UDP, in order to ensure that any node that can communicate over TCP but has been isolated from UDP traffic has a chance to respond. This mechanism can be disabled, but that was not found to reduce the failure detection time.

7. Once the probe interval timer runs out and no acknowledgment has been received, the node is marked as suspect and the process moves on to the suspicion stage.

In the case of 2 nodes only, this means that the time in which failure can be detected with regard to this stage is fairly predictable, at *probe interval* units of time. This is due to the fact that if the node is dead, the algorithm waits until the probe interval timer has run out before moving on, with no possibility of the process being ended earlier. On the other hand, with more than 2 nodes, the maximum time remains the same, but the minimum can be lower, as the indirect probes can end the process earlier by confirming that the node is indeed unresponsive.

4.2.5 Suspicion Stage

The suspicion stage is fairly straightforward - it is essentially extra time being given to the probed node to refute the suspicion that it has failed. It is possible to eliminate this stage altogether by setting the suspicion multiplier to 0, which was the suitable choice for this project, considering that its focus is on minimizing the failure detection time. With this in mind, the suspicion stage has no impact on the failure detection times recorded in this project, but can be configured to be included in the process, if deemed necessary.

Eliminating the suspicion stage altogether could potentially introduce false positives with regard to failure detection, but due to the focus on maximizing the speed at which failure is detected, this was considered an acceptable tradeoff.

4.2.6 Expected Failure Detection Time

Due to the crucial and sensitive nature of mission-critical systems, it is important to know the expected values of any failure detection mechanism, as well as to be able to explain how these expected times are reached. In this section, only scenarios with exactly 2 nodes are considered.

As explained in the previous sections, the probing and suspicion stages are in practice constant - the former always takes *probe interval* units of time to complete, while the latter is essentially nonexistent, provided that the suspicion multiplier parameter is set to 0, as is the case for the optimized variant of the K8s-and-Serf implementation.

The selection stage, on the other hand, can fluctuate. A best- and worst-case scenario can be identified, both of which are represented in the timeline seen in Figure 4.3. The two key factors that determine the failure detection time are timing with respect to the time in-between probes, as well as whether a probing round will be wasted due to the variable incrementing issue described in Section 4.2.5.

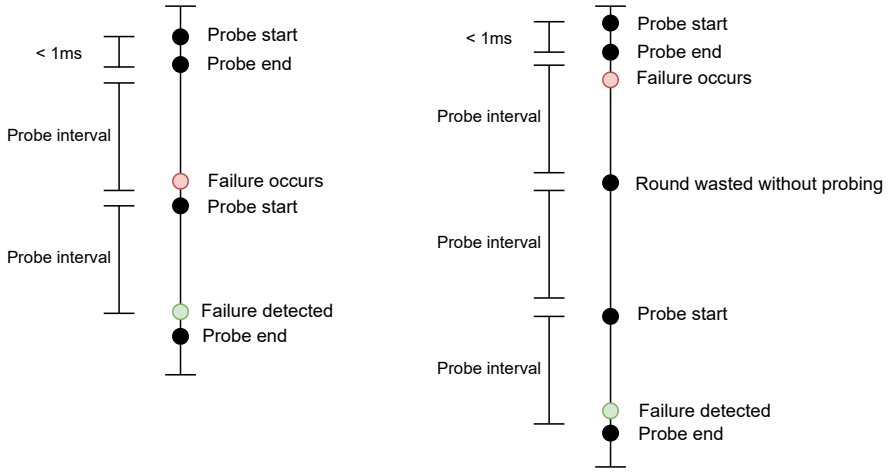

FIGURE 4.3: The approximate timeline for the best- and worst-case scenario for the selection stage. The former is depicted on the left side of the diagram, the latter on the right.

In the best-case scenario, the wasted probing round does not occur right after a node fails, and so no extra *probe interval* units of time are added to the time. Furthermore, the timing of the failure coincides with the end of the interval that occurs between probes, ensuring that the failure is detected nearly immediately. As such, in the best case scenario failure is detected within approximately *probe interval* units of time, caused by the probing stage of the process.

In the worst-case scenario, the opposite conditions can be observed. The failure occurs near the start of the interval between probes, causing an extra *probe interval* unit of time to elapse without the failure being detected. Additionally, after the interval passes, the next probing round is wasted due to the variable incrementing issue, adding another *probe interval* unit of time due to the interval that must occur between probes. Adding this extra

time to that of the fixed time of the probing stage, the worst-case scenario is approximately $3 * probe\ interval$.

4.3 Architecture

Two types of experimental environments were set up - one that simulates the system of Thales as managed by Kubernetes only (from now on referred to as K8s-only), and one that incorporates Serf within Kubernetes (K8s-and-Serf). A diagram depicting the two configurations can be seen in Figures 4.4 and 4.5, respectively.

First, the aspects common to both experimental implementations will be discussed.

In order to simulate a distributed system with multiple nodes, without having the access to physical machines to do so, *Kubernetes in Docker* [9], also known as *kind*, is used. The number of exact number of nodes varies depending on the type of experiment and can be easily configured to modify the scenario. However, limitations were encountered with regard to the number of nodes, explained in more detail in Section 4.5.

Furthermore, the *Cyclone DDS* implementation of the DDS protocol, in particular its Python API, is utilized for establishing the communication between the different components of the system. More details on this aspect can be found in Section 2.6. Unless configured otherwise, DDS operates via multicast, which is not supported by Kubernetes's default CNI. As such, the Weave Net CNI was used instead, allowing multicast traffic on the host network using Kubernetes.

In the implemented architecture, two distinct types of applications can be defined - *publishers*, which can continuously publish mock sensor data to DDS, as well as *subscribers*, which continuously receive any data that has been published to DDS. The number of publishers varies depending on the type of implementation, and is controlled by a Deployment resource to ensure flexibility. Subscribers, on the other hand, are placed on every node using a DaemonSet resource for monitoring purposes. Furthermore, publishers send out data every 10 milliseconds to ensure accuracy, but this can be adjusted when necessary, for example to improve readability of the logs.

Each DDS participant is assigned an ID via UUID4, as no unique identifier is provided by Cyclone DDS. Sensor data is of the following format:[*timestamp received*] - Participant [*participant ID*] has received message: [*sensor data*] at [*timestamp sent*]. Timestamps are of the format [*YYYY-MM-DD HH:MM:SS.MS*]. The aim is to be able to keep precise track of the duration of the failure, down to the milliseconds.

The two experimental implementations differ in multiple key ways. Aspects unique to the K8s-only implementation include:

- There is only one replica of the publisher application running at a time. Upon failure, Kubernetes restarts the container in order to return to normal operation. This means that warm standby is *not* used as the failover approach in this scenario, but rather cold standby. This design choice is an approximation of a system that uses Kubernetes only for managing failure detection, as there is no existing approach for implementing warm standby in Kubernetes.
- By default, it takes 5 minutes (300s) for Kubernetes to remove a Pod from a Node once the node fails. This was changed to 5s, even when measuring the performance with the default settings.

- The optimized version of the K8s-only system is inspired by [57] and cross-referenced with the Kubernetes documentation [8]. The modified elements are:
  - *node-status-update-frequency* - indicates how often the status of the node is posted to the API. The default value is 10s, reduced to 4s.
  - *node-monitor-period* - how often the API server is queried regarding the status of each node. The default value is 5s, reduced to 2s.
  - *node-monitor-grace-period* - the duration for which a suspected node is allowed to be unresponsive before being marked as failed. The default value is 50s, reduced to 16s for the optimized version.

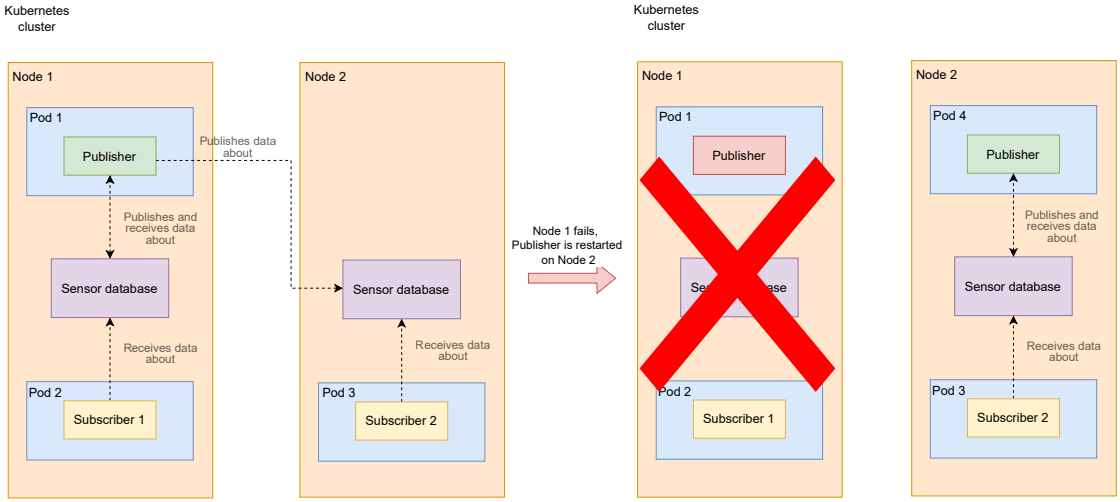

FIGURE 4.4: Diagram of the system using Kubernetes only, showing the state in normal operation and after failure.

Aspects unique to the K8s-and-Serf implementation, on the other hand, include:

- Warm standby is incorporated instead of cold standby, compared to the K8s-only solution. There are multiple replicas of the publisher application, only one of which is active. The rest are in an idle state, ready to take over when the primary replica fails. Until it does, they are only subscribed to the topic in order to be as up to date as possible in preparation for failover. It is worth noting that at the moment, this aspect is not being utilized due to Cyclone DDS's lack of support for transient and persistent storage (see Section 2.6.1 for more information on this issue).
- Each publisher, regardless of its status as active or idle, is monitored by a Serf agent. This is the key distinction between the two implementations and it is the primary failure detection mechanism employed in this implementation. Serf is responsible for the health monitoring and handling of failure across the replicas.
- Only the publishers are monitored by Serf agents, as it is unnecessary for these experiments for the subscribers to be monitored for failure. Another reason for this decision is that it allows for more clarity and reduced noise when analyzing the Serf algorithm.
- Serf is running at the container level for each instance of the application.
- Serf is exposed on the network by a Service resource.

- The distinction between active and idle is implemented via the use of Serf's tagging and event handler capabilities. Initially, all instances behave as if they are idle. Each publisher, irregardless of its intended role, periodically checks for whether its role has been set to active. Once all Pods are ready, the first publisher as listed by Kubernetes is marked as active, the second as idle. As soon as failure occurs, a specific failover script is triggered in all publishers (barring the failed one) that allows for the replica marked as idle to change its role to active, thus becoming an active publisher.
- The optimized version of the K8s-and-Serf implementation involves modifying select parameters, in particular in its library *memberlist*, details for which are laid out in Section 4.2.1. Discussion on the values of the parameters that were explored during the research can be found in Section 4.4. More specifically, the relevant parameters and their final chosen values are:
  - *Suspicion multiplier* - 0, changed from 2.
  - *Probe interval* - 100 milliseconds, changed from 1 second.
  - *Probe timeout* - 50 milliseconds, changed from 500 milliseconds.

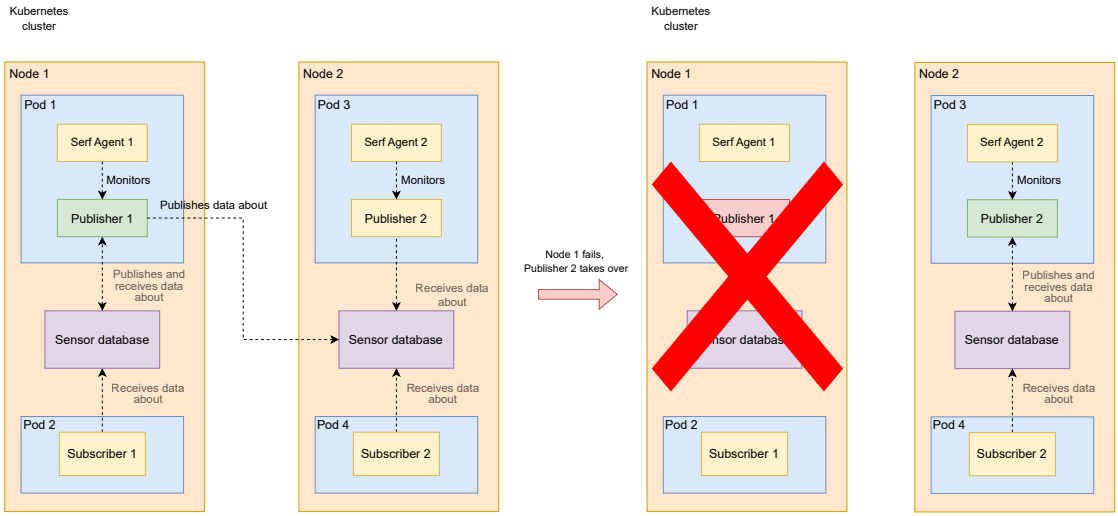

FIGURE 4.5: Diagram of the system using Kubernetes in combination with Serf, showing the state in normal operation and after failure.

Once the experiments are ran and metrics are collected for both implementations, a comparison is made based on the failure detection time. The results of these experiments can be seen in Section 4.4.

4.4 Results

In this section, the results obtained from running the experiments for both the K8s-only and K8s-and-Serf implementations are displayed and the findings derived from them are discussed, providing an answer to RQ2b.

In order to be able to accurately measure the failover time, the following metrics were collected, in the form of timestamps:

- *Start of failure* - the timestamp of the last message received by subscribers before the node fails. More specifically, the publisher sends as part of their message the times-

tamp at which they sent the specific message. Said timestamp is the one considered, in order to ensure maximum accuracy in the milliseconds.

- *Failure detected* - the timestamp at which the failure was detected:
  - For the K8s-only solution, this is the timestamp at which the node is marked as NotReady.
  - For the K8s-and-Serf solution, this is the timestamp of the EventMemberFailed event, as announced by Serf.
- *End of failure* - the timestamp at which the first message after the failure is received by subscribers.
- *Failure detection time* - failure detected − start of failure. The most significant metric for this research, since it is strictly consistent for both types of implementations, as well as being completely independent from any design choices of the application implementations. Instead, it is dependent solely on the two failure detection mechanisms that are being compared - Kubernetes and Serf.
- *Total restart time* - end of failure − failure detected.
- *Total failover time* - end of failure − start of failure.

Aside from the author's personal device, the experiments were also attempted using Thales's more powerful machines, but no particular difference in results could be observed. The experiments were also ran on a virtual machine on both hosts, again with no particular difference in the outcome. Based on this information, it can be concluded that the failure detection performance is not notably affected by hardware capabilities, but rather by the configuration and the design of the system.

For each type of experiment, 20 samples were collected for a total of 80 samples, and their averages are depicted in seconds in Table 4.1. 20 samples per experiment was deemed enough as the standard deviation between samples is quite small, while the effect size between the different implementation is fairly large. The full results can be seen in Appendix A.

Implementation	Configuration	Failure Detection Time	Restart Time	Total Time
K8s-only	Default	31.484	07.512	38.997
	Optimized	10.621	04.443	15.064
K8s-and-Serf	Default	06.194	**01.044**	07.238
	Optimized	**00.218**	**01.044**	**01.261**

TABLE 4.1: Average results in seconds of the experiments for K8s-only and K8s-and-Serf, with both the default and optimized settings of each.

To start with, given that the K8s-only implementation uses cold standby, while the K8s-and-Serf implementation uses warm standby, the two cannot reasonably be compared with regard to restart time. It is inevitable that the K8s-and-Serf experiments will have a faster outcome on those metrics, as the component that will take over after failure is already running and ready to begin publishing data before failure even occurs, while in the K8s-only experiments, the failed component needs to be restarted.

The results indicate this as well - as expected, the restart time is considerably faster for the K8s-and-Serf implementation (1.044s) compared to the K8s-only implementation (7.512s for the default version, 4.443s for the optimized). Because of the difference in architecture design, this metric is not indicative of whether Serf is an improvement over Kubernetes alone, but it further supports the idea that the use of warm standby is preferable to the use of cold standby.

The key metric on which it can be determined whether the addition of Serf is an improvement is the failure detection time. The first and most crucial observation is in the difference between the K8s-only and the K8s-and-Serf implementation. It is immediately clear that the use of Serf, even in with its default configuration, significantly improves the failure detection time compared to either of the K8s-only configurations. The K8s-only version at worst averages a failure detection time of 31.484s, and at its most improved, it can achieve a time of 10.621s on average. K8s-and-Serf, on the other hand, has a failure detection time of 6.194s at worst, and 0.218s at its most optimized, achieving a failure detection time in the milliseconds. Furthermore, these results show an improvement over the capabilities of Thales's current system.

4.4.1 Multiple Nodes

Despite not being the main focus of this research, the experiments were also attempted on a smaller scale with more than 2 nodes for the K8s-and-Serf implementation in order to observe any changes in the behaviour of the system, with the aim of answering RQ2c.

The only difference between implementations with 2 nodes and those with more than 2 nodes appears to be the use of indirect checks in the probing stage, immediately after the direct probe fails by way of the probe timeout timer running out. As a small case example, experiments of 5 samples each were run on an environment with 5 nodes, with the *indirect checks* parameter set to 2, 3 and 4 in order to observe the behaviour. The outcome of these test runs can be seen in Table 4.2.

Indirect Checks	Failure Detection Time	Restart Time	Total Time
2	0:00:00.188	0:00:01.060	0:00:01.247
3	0:00:00.177	0:00:01.049	0:00:01.226
4	0:00:00.202	0:00:01.049	0:00:01.252
Averages	**0:00:00.189**	0:00:01.053	0:00:01.242

TABLE 4.2: Average results of running the experiments for an environment with 5 nodes and different values for *indirect check*.

In theory, the minimum and maximum expected time remains the same as for implementations with only 2 nodes. However, it can be observed that the failure detection time is slightly faster than the average observed for 2 nodes only. This is due to the presence of indirect checks, which can confirm that the node has failed faster and interrupt the waiting time for the *probe interval* timer within the probing stage.

As such, it can be concluded that although the minimum and maximum expected failure detection time is still the same, the use of more than 2 nodes, and therefore indirect checks, can speed up the process by some milliseconds, in this case by approximately 30 milliseconds.

4.5 Limitations

In this section, factors that likely had a limiting impact on this research are discussed.

An important limitation that had impact on the scope of this research was the lack of access to physical nodes or hardware that could support the simulation of multiple nodes. Attempting to use more than 6 nodes with Kubernetes in Docker on the available machines proved to be impossible, thus limiting the original intent of exploring the effect of using Kubernetes and Serf on a larger number of nodes. As a result, these scenarios were explored on a smaller scale and only to observe any difference in behaviour compared to the use of only 2 nodes.

The scope of this research was further reduced by CycloneDDS's lack of support for persistent and transient storage at the time of writing [1], as the original intent was also to have continuity between the content of the messages before and after the point of failure. Although this aspect was always secondary, it is nevertheless a limitation on the original intent.

Another limiting aspect was the memberlist library's built-in constraints on the relevant parameters during the build process. Due to these constraints, this research utilized the lowest possible values to which they could be set, but it is likely that with more freedom for the use case of this project, the failure detection time could have been lowered further.

With regard to the accuracy of simulating a mission-critical system, the system implemented for the purposes of this project likely does not fully reflect a mission-critical product used in practice. Due to confidentiality reasons, access to Thales's systems was not granted for this project.

4.6 Recommendations

In this section, a discussion on starting off points for future research are presented, based on the scope limitations and results of this project.

To start with, more extensive experiments could be performed on environments with more than 2 nodes, given sufficiently powerful hardware or access to physical nodes. This could extend the work done in this research by examining whether there is a point at which the K8s-and-Serf solution becomes less efficient at failure detection, for example.

Another possibility could be the exploration of a custom solution designed specifically for optimized failure detection in a distributed mission-critical system. Considering that the memberlist library imposes limitations on how much the parameters can be tuned, likely due to other scenarios in which failure detection should not be prioritized, it is highly likely that a more optimal solution can be found by implementing an algorithm designed purely for optimized failure detection. This is especially worthwhile for mission-critical systems that expect a failure detection time in the microseconds. Furthermore, Serf's expected failure detection time is a range, rather than a fixed value, and it may not be predictable enough to suit the needs of mission-critical systems, which is another problem for which a custom solution could be beneficial.

Additionally, the restart times of this implementation could most likely be improved. The restart time is influenced by the means by which the experiments were implemented, and considering that that aspect was not the focus of this research, it is highly likely that the restart times are not optimal.

Chapter 5

Conclusion

In this thesis, the problems associated with achieving HA in the context of distributed mission-critical systems using Kubernetes are established. To start with, three possible solutions to the problem of fast failover are compared, namely Consul, ZooKeeper and Serf. The aspects that are considered are those most relevant to the specific challenges in the established context, in particular the use of quorums, centralization, consistency type, Kubernetes compatibility, scalability and ease of use.

Through that comparison, it is clear that the most promising alternative to use for addressing the established problem would be Serf. This is due to the fact that Serf does not present the same existing drawbacks that Consul and ZooKeeper have, such a quorum that is incompatible with the topologies of many mission-critical systems, or centralization and strong consistency that limit the system's availability.

Based on these findings, an experimental implementation was created which approximates that of a distributed mission-critical system such as Thales's with its use of DDS for communication between publishers and subscribers and a simulation of multiple physical nodes. In this environment, failure is introduced to the node of the active publisher and the failure detection time is recorded. This process is done for a variant of the implementation that works with Kubernetes only, as well as one that works with Kubernetes and Serf in combination. Both of these variants are further optimized and compared.

The findings of this research strongly indicate that the use of Serf, even without optimization, greatly improves the failure detection time in a distributed system, as it is capable of reaching a time as low as 0.22 seconds, compared to the failure detection time of Kubernetes even at its best, at approximately 10.62 seconds.

Additionally, the failure detection time can be further reduced to approximately 0.19 seconds with Serf, provided that more than 2 nodes are used in the environment. This is due to the fact that indirect checks generally speed up the failure detection process, but they require nodes other than the failed node and the node performing the probe.

It is also possible that a custom solution could achieve even better results, which is a topic worth exploring in further research, especially if an organization requires a failure detection time in the microseconds, or for the solution to be less complex in order to ensure a fixed estimated failure detection time. Regardless, it can be concluded from the results of this research that mission-critical organizations are highly likely to benefit from incorporating Serf into their Kubernetes-managed distributed systems.

Bibliography

[1] Question on durability:persistence. URL: https://github.com/eclipse-cyclonedds/cyclonedds/issues/2054.

[2] Diogo A. B. Fernandes, Liliana F. B. Soares, João V. Gomes, Mario Freire, and Pedro R. M. Inácio. Security issues in cloud environments - a survey. *International Journal of Information Security*, apr 2013.

[3] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Kubernetes as an availability manager for microservice applications. *Journal of Network and Computer Applications*, 2019.

[4] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 2019.

[5] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. A kubernetes controller for managing the availability of elastic microservice based stateful applications. *Journal of Systems and Software*, 175, may 2021.

[6] Shahbaz Afzal and Kavitha Ganesh. Load balancing in cloud computing – a hierarchical taxonomical classification. *Journal of Cloud Computing*, 2019.

[7] Inc. Amazon Web Services. What is distributed computing? URL: https://aws.amazon.com/what-is/distributed-computing/.

[8] The Kubernetes Authors. Kubernetes. URL: https://kubernetes.io/.

[9] The Kubernetes Authors. kind, 2024. URL: https://kind.sigs.k8s.io/.

[10] Amazon Web Services (AWS). What is virtualization? - cloud computing virtualization explained. URL: https://aws.amazon.com/what-is/virtualization/.

[11] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1, oct 2017.

[12] Marco Barletta, Marcello Cinque, Catello Di Martino, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Mutiny! how does kubernetes fail, and what can we do about it?, 2024. arXiv:2404.11169.

[13] David Bermbach, Liang Zhao, and Sherif Sakr. Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. In *Performance Characterization and Benchmarking*, pages 32–47, 2013.

[14] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, sep 2014.

[15] Bharat K. Bhargava, Rohit Ranchal, and Pelin Angin. Privacy-preserving data sharing and adaptable service compositions in mission-critical clouds. In *International Symposium on Intelligent Control*, 2021. URL: https://api.semanticscholar.org/CorpusID:232401060.

[16] Ken Birman. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review*, 41(5):8–13, oct 2007.

[17] Eric Brewer. Cap twelve years later: How the 'rules' have changed. *Computer*, 45(2):23–29, feb 2012.

[18] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and María S. Pérez. Consistency in the cloud: When money does matter! In *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2013.

[19] Cisco. What is high availability? URL: https://www.cisco.com/c/en/us/solutions/hybrid-work/what-is-high-availability.html.

[20] Google Cloud. What is kubernetes? URL: https://cloud.google.com/learn/what-is-kubernetes.

[21] Marcin Copik, Alexandru Calotoiu, Pengyu Zhou, Konstantin Taranov, and Torsten Hoefler. Faaskeeper: Learning from building serverless services with zookeeper as an example. In *33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, may 2024.

[22] Paul Cormier. The state of enterprise open source: A red hat report. Technical report, Red Hat, feb 2022.

[23] Abhinandan Das, Indranil Gupta, and Ashish Motivala. SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 303 – 312. IEEE, 2002.

[24] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, page 195–216. Springer Cham, 2017.

[25] Scott Eisele, Istvan Mardari, Abhishek Dubey, and Gabor Karsai. Riaps: Resilient information architecture platform for decentralized smart systems. In *IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, may 2017.

[26] etcd Authors. etcd. URL: https://etcd.io/.

[27] Apache Software Foundation. Apache zookeeper. URL: https://cwiki.apache.org/confluence/display/ZOOKEEPER.

[28] The Apache Software Foundation. Zookeeper overview. URL: https://zookeeper.apache.org/doc/r3.5.1-alpha/zookeeperOver.html.

[29] Seth Gilbert and Nancy A. Lynch. Perspectives on the cap theorem. *Computer*, 45:30–36, feb 2012.

[30] Google. Google scholar. URL: https://scholar.google.com/intl/en/scholar/about.html.

[31] Albert Gustavsson. A comparative evaluation of failover mechanisms for mission-critical financial applications in public clouds. Master's thesis, Umeå University, 2023.

[32] HashiCorp. Consul. URL: https://www.consul.io/.

[33] Hashicorp. memberlist. URL: https://github.com/hashicorp/memberlist.

[34] HashiCorp. Recommendations for operating consul at scale. URL: https://developer.hashicorp.com/consul/docs/architecture/scale.

[35] HashiCorp. Serf. URL: https://www.serf.io/.

[36] Hevner, R. Alan, March, T. Salvatore, and Sudha. Design Science in Information Systems Research. *Management Information Systems Quarterly*, 28:75—106, March 2004. URL: https://www.researchgate.net/publication/201168946_Design_Science_in_Information_Systems_Research.

[37] Mike Hinchey and Lorcan Coyle. Evolving critical systems: A research agenda for computer-based systems. In *International Conference and Workshop on Engineering of Computer-Based Systems*, 2010.

[38] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010. URL: https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems.

[39] Zhuangwei Kang, Kyoungho An, Aniruddha Gokhale, and Paul Pazandak. A comprehensive performance evaluation of different kubernetes cni plugins for edge-based and containerized publish/subscribe applications. In *IEEE International Conference on Cloud Engineering*, 2021.

[40] John C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, page 547–550. Association for Computing Machinery, 2002. URL: https://doi.org/10.1145/581339.581406.

[41] Marta Kwiatkowska, Gethin Norman, and David Parker. Analysis of a gossip protocol in prism. *ACM SIGMETRICS Performance Evaluation Review*, 36(3):17–22, nov 2008.

[42] Gregory Levitin, Liudong Xing, and Yuanshun Dai. Cold vs. hot standby mission operation cost minimization for 1-out-of-n systems. *European Journal of Operational Research*, 234(1):155–162, apr 2014.

[43] Fei-fei Li, Xiang-zhan Yu, and Gang Wu. Design and implementation of high availability distributed system based on multi-level heartbeat protocol. In *International Conference on Control, Automation and Systems Engineering, CASE*, 2009.

[44] Mina Nabi, Maria Toeroe, and Ferhat Khendek. Availability in the cloud: State of the art. *Journal of Network and Computer Applications*, 60:54–67, jan 2016.

[45] Object Management Group (OMG). Dds portal - data distribution services. URL: https://www.dds-foundation.org/.

[46] Object Management Group (OMG). Data distribution service for real-time systems, jan 2007. URL: https://www.omg.org/spec/DDS/1.2/PDF/.

[47] Andre Oriani and Islene C. Garcia. From backup to hot standby: High availability for hdfs. *31st International Symposium on Reliable Distributed Systems*, 2012.

[48] Ashish Patel. Kubernetes — Architecture and Cluster Components Overview | DevOps Mojo. *Medium*, April 2024. URL: https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd.

[49] Neena Pemmaraju and Cody DeArkland. Introduction to consul 1.9 and a demo, 2020. URL: https://www.hashicorp.com/fr/resources/keynote-consul-overview-and-1-9-demo.

[50] E.G. Radhika and G. Sudha Sadasivam. A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment. In *Materials Today: Proceedings*, 2021.

[51] Cristian Ramon-Cortes, Pol Alvarez, Francesc Lordan, Javier Alvarez, Jorge Ejarque, and Rosa M. Badia. A survey on the distributed computing stack. *Computer Science Review*, 42, 2021.

[52] Max Riesewijk. High availability orchestration of linux containers in mission-critical on-premise systems. Master's thesis, University of Twente, 2020.

[53] Open Robotics. Dds implementations. URL: https://docs.ros.org/en/humble/Installation/DDS-Implementations.html.

[54] Joe Stubbs, Walter Moreira, and Rion Dooley. Distributed systems of microservices using docker and serfnode. In *International Workshop on Science Gateways (IWSG)*, 2015.

[55] dpotman thijsmie, eboasson. Python binding for eclipse cyclone dds. URL: https://github.com/eclipse-cyclonedds/cyclonedds-python.

[56] Maria Toeroe and Francis Tam. *Service Availability: Principles and Practice*. Wiley, 2012.

[57] v1ktoor. Improving kubernetes reliability: quicker detection of a node down, 2016. URL: https://fatalfailure.wordpress.com/2016/06/10/improving-kubernetes-reliability-quicker-detection-of-a-node-down/.

[58] Marko Vukolic. The origin of quorum systems. *Bull. EATCS*, 101:125–147, 2010. URL: https://api.semanticscholar.org/CorpusID:43151159.

[59] Roel Johannes Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, Berlin, Germany, 2014. URL: https://link.springer.com/book/10.1007/978-3-662-43839-8.

[60] Jiuping Xu and Lei Xu. System assessment. In *Integrated System Health Management*, pages 201–245. Academic Press, 2017.

[61] Hyunsik Yang and Younghan Kim. Design and implementation of fast fault detection in cloud infrastructure for containerized iot services. *Sensors*, 2020.

Appendix A

Experiment Data

Sample #	Failure Start	Failure Detected	Failure End	Total Detection Time	Total Restart Time	Total Time
1	14:57:31.617	14:57:58.000	14:58:05.130	0:00:26.383	0:00:07.130	0:00:33.513
2	15:03:46.635	15:04:19.000	15:04:26.753	0:00:32.365	0:00:07.753	0:00:40.118
3	15:07:18.657	15:07:53.000	15:08:00.526	0:00:34.343	0:00:07.526	0:00:41.869
4	15:11:50.972	15:12:26.000	15:12:33.908	0:00:35.028	0:00:07.908	0:00:42.936
5	15:15:49.225	15:16:20.000	15:16:27.675	0:00:30.775	0:00:07.675	0:00:38.450
6	15:18:58.912	15:19:29.000	15:19:36.720	0:00:30.088	0:00:07.720	0:00:37.808
7	15:25:18.853	15:25:50.000	15:25:57.757	0:00:31.147	0:00:07.757	0:00:38.904
8	15:28:35.483	15:29:09.000	15:29:16.008	0:00:33.517	0:00:07.008	0:00:40.525
9	15:31:39.851	15:32:07.000	15:32:14.320	0:00:27.149	0:00:07.320	0:00:34.469
10	15:35:12.156	15:35:48.000	15:35:55.383	0:00:35.844	0:00:07.383	0:00:43.227
11	15:39:05.728	15:39:39.000	15:39:46.267	0:00:33.272	0:00:07.267	0:00:40.539
12	11:45:01.347	11:45:34.000	11:45:41.540	0:00:32.653	0:00:07.540	0:00:40.193
13	11:48:40.718	11:49:14.000	11:49:21.390	0:00:33.282	0:00:07.390	0:00:40.672
14	11:52:00.265	11:52:27.000	11:52:34.029	0:00:26.735	0:00:07.029	0:00:33.764
15	11:55:12.976	11:55:47.000	11:55:54.597	0:00:34.024	0:00:07.597	0:00:41.621
16	11:58:30.524	11:59:00.000	11:59:07.348	0:00:29.476	0:00:07.348	0:00:36.824
17	12:09:50.028	12:10:18.000	12:10:25.718	0:00:27.972	0:00:07.718	0:00:35.690
18	12:13:08.261	12:13:39.000	12:13:46.766	0:00:30.739	0:00:07.766	0:00:38.505
19	12:16:21.941	12:16:55.000	12:17:02.602	0:00:33.059	0:00:07.602	0:00:40.661
20	12:19:37.162	12:20:09.000	12:20:16.808	0:00:31.838	0:00:07.808	0:00:39.646
Averages				**0:00:31.484**	**0:00:07.512**	**0:00:38.997**

TABLE A.1: Average results in seconds of the experiments for K8s-only with default settings.

Sample #	Failure Start	Failure Detected	Failure End	Total Detection Time	Total Restart Time	Total Time
1	12:31:23.156	12:31:34.000	12:31:38.375	0:00:10.844	0:00:04.375	0:00:15.219
2	12:34:32.376	12:34:41.000	12:34:45.456	0:00:08.624	0:00:04.456	0:00:13.080
3	12:37:08.754	12:37:20.000	12:37:24.814	0:00:11.246	0:00:04.814	0:00:16.060
4	12:40:03.570	12:40:15.000	12:40:19.474	0:00:11.430	0:00:04.474	0:00:15.904
5	12:50:30.598	12:50:38.000	12:50:42.730	0:00:07.402	0:00:04.730	0:00:12.132
6	12:59:28.537	12:59:40.000	12:59:44.731	0:00:11.463	0:00:04.731	0:00:16.194
7	13:02:14.474	13:02:28.000	13:02:32.170	0:00:13.526	0:00:04.170	0:00:17.696
8	13:05:14.138	13:05:24.000	13:05:28.171	0:00:09.862	0:00:04.171	0:00:14.033
9	13:08:05.534	13:08:16.000	13:08:20.278	0:00:10.466	0:00:04.278	0:00:14.744
10	13:10:47.318	13:10:56.000	13:11:00.532	0:00:08.682	0:00:04.532	0:00:13.214
11	13:13:25.745	13:13:34.000	13:13:38.598	0:00:08.255	0:00:04.598	0:00:12.853
12	13:16:01.346	13:16:12.000	13:16:16.521	0:00:10.654	0:00:04.521	0:00:15.175
13	13:18:57.687	13:19:10.000	13:19:14.332	0:00:12.313	0:00:04.332	0:00:16.645
14	13:21:41.403	13:21:52.000	13:21:56.237	0:00:10.597	0:00:04.237	0:00:14.834
15	13:24:23.976	13:24:35.000	13:24:39.771	0:00:11.024	0:00:04.771	0:00:15.795
16	13:27:34.764	13:27:46.000	13:27:50.057	0:00:11.236	0:00:04.057	0:00:15.293
17	13:30:09.595	13:30:22.000	13:30:26.291	0:00:12.405	0:00:04.291	0:00:16.696
18	13:32:49.298	13:33:00.000	13:33:04.067	0:00:10.702	0:00:04.067	0:00:14.769
19	13:35:21.637	13:35:31.000	13:35:35.832	0:00:09.363	0:00:04.832	0:00:14.195
20	13:37:59.674	13:38:12.000	13:38:16.426	0:00:12.326	0:00:04.426	0:00:16.752
Averages				**0:00:10.621**	**0:00:04.443**	**0:00:15.064**

TABLE A.2: Average results in seconds of the experiments for K8s-only with optimized settings.

Sample #	Failure Start	Failure Detected	Failure End	Total Detection Time	Total Restart Time	Total Time
1	15:00:10.718	15:00:17.167	15:00:18.201	0:00:06.449	0:00:01.034	0:00:07.483
2	15:06:14.498	15:06:20.891	15:06:21.939	0:00:06.393	0:00:01.048	0:00:07.441
3	15:09:44.557	15:09:50.536	15:09:51.579	0:00:05.979	0:00:01.043	0:00:07.022
4	15:13:22.181	15:13:27.797	15:13:28.833	0:00:05.616	0:00:01.036	0:00:06.652
5	15:16:54.815	15:17:01.758	15:17:02.794	0:00:06.943	0:00:01.036	0:00:07.979
6	15:20:14.471	15:20:20.809	15:20:21.864	0:00:06.338	0:00:01.055	0:00:07.393
7	15:27:15.017	15:27:21.564	15:27:22.609	0:00:06.547	0:00:01.045	0:00:07.592
8	15:30:48.743	15:30:55.140	15:30:56.189	0:00:06.397	0:00:01.049	0:00:07.446
9	15:34:41.488	15:34:47.943	15:34:48.980	0:00:06.455	0:00:01.037	0:00:07.492
10	15:38:09.062	15:38:15.860	15:38:16.895	0:00:06.798	0:00:01.035	0:00:07.833
11	15:41:49.983	15:41:56.182	15:41:57.228	0:00:06.199	0:00:01.046	0:00:07.245
12	15:45:15.212	15:45:20.446	15:45:21.485	0:00:05.234	0:00:01.039	0:00:06.273
13	15:48:46.330	15:48:52.135	15:48:53.181	0:00:05.805	0:00:01.046	0:00:06.851
14	15:51:59.458	15:52:05.733	15:52:06.777	0:00:06.275	0:00:01.044	0:00:07.319
15	15:55:15.284	15:55:21.859	15:55:22.898	0:00:06.575	0:00:01.039	0:00:07.614
16	15:58:31.202	15:58:36.657	15:58:37.695	0:00:05.455	0:00:01.038	0:00:06.493
17	16:01:57.621	16:02:03.339	16:02:04.384	0:00:05.718	0:00:01.045	0:00:06.763
18	16:05:11.058	16:05:17.686	16:05:18.752	0:00:06.628	0:00:01.066	0:00:07.694
19	16:08:30.367	16:08:35.694	16:08:36.740	0:00:05.327	0:00:01.046	0:00:06.373
20	16:12:04.901	16:12:11.650	16:12:12.697	0:00:06.749	0:00:01.047	0:00:07.796
Averages				**0:00:06.194**	**0:00:01.044**	**0:00:07.238**

TABLE A.3: Average results in seconds of the experiments for K8s-and-Serf with default settings.

Sample #	Failure Start	Failure Detected	Failure End	Total Detection Time	Total Restart Time	Total Time
1	13:58:09.299	13:58:09.604	13:58:10.650	0:00:00.305	0:00:01.046	0:00:01.351
2	14:02:00.066	14:02:00.303	14:02:01.357	0:00:00.237	0:00:01.054	0:00:01.291
3	14:05:34.197	14:05:34.382	14:05:35.417	0:00:00.185	0:00:01.035	0:00:01.220
4	14:09:09.122	14:09:09.250	14:09:10.293	0:00:00.128	0:00:01.043	0:00:01.171
5	14:12:28.172	14:12:28.449	14:12:29.498	0:00:00.277	0:00:01.049	0:00:01.326
6	14:15:43.280	14:15:43.554	14:15:44.594	0:00:00.274	0:00:01.040	0:00:01.314
7	14:19:03.418	14:19:03.529	14:19:04.571	0:00:00.111	0:00:01.042	0:00:01.153
8	14:22:21.322	14:22:21.573	14:22:22.615	0:00:00.251	0:00:01.042	0:00:01.293
9	14:25:39.582	14:25:39.865	14:25:40.906	0:00:00.283	0:00:01.041	0:00:01.324
10	14:28:56.470	14:28:56.728	14:28:57.775	0:00:00.258	0:00:01.047	0:00:01.305
11	14:32:14.001	14:32:14.130	14:32:15.169	0:00:00.129	0:00:01.039	0:00:01.168
12	14:35:32.411	14:35:32.564	14:35:33.611	0:00:00.153	0:00:01.047	0:00:01.200
13	14:38:50.965	14:38:51.221	14:38:52.268	0:00:00.256	0:00:01.047	0:00:01.303
14	14:42:10.385	14:42:10.523	14:42:11.562	0:00:00.138	0:00:01.039	0:00:01.177
15	14:45:29.906	14:45:30.169	14:45:31.215	0:00:00.263	0:00:01.046	0:00:01.309
16	14:48:49.757	14:48:50.002	14:48:51.059	0:00:00.245	0:00:01.057	0:00:01.302
17	14:52:07.707	14:52:07.974	14:52:09.011	0:00:00.267	0:00:01.037	0:00:01.304
18	14:55:27.391	14:55:27.570	14:55:28.607	0:00:00.179	0:00:01.037	0:00:01.216
19	14:58:44.375	14:58:44.590	14:58:45.627	0:00:00.215	0:00:01.037	0:00:01.252
20	15:02:04.493	15:02:04.690	15:02:05.735	0:00:00.197	0:00:01.045	0:00:01.242
Averages				**0:00:00.218**	**0:00:01.044**	**0:00:01.261**

TABLE A.4: Average results in seconds of the experiments for K8s-and-Serf with optimized settings.