TU WIEN Informatics

Unterschiede in fehlertoleranten Methoden für zustandsbehaftete container-basierte Fog Applikationen

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software and Information Engineering

eingereicht von

Alexander Gschnitzer
Matrikelnummer 01652750

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Projektass. Dipl.-Ing. Patrick Denzler, MSc, MBA
Projektass. Dipl.-Ing. Thomas Preindl, BSc

Wien, 20. Juli 2022

Alexander Gschnitzer	Wolfgang Kastner

TU WIEN Informatics

Differences in fault-tolerant methods in stateful container-based fog applications

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Alexander Gschnitzer
Registration Number 01652750

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Projektass. Dipl.-Ing. Patrick Denzler, MSc, MBA
Projektass. Dipl.-Ing. Thomas Preindl, BSc

Vienna, 20th July, 2022

Alexander Gschnitzer | Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Alexander Gschnitzer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. Juli 2022

Alexander Gschnitzer

Danksagung

Ich möchte meinen Betreuern Thomas Preindl und Patrick Denzler für ihre Beratung und durchgängige Unterstützung während meiner Arbeit danken. Sie haben mir mit ihren nützliche Einblicken und Rückmeldungen, vor allem während der Entwicklungsphase dieser Arbeit geholfen, meine Ideen zu konkretisieren. Die regelmäßigen Diskussionen über Fehlertoleranz in zustandsbehafteten Anwendungen, verschiedenen Herangehensweisen und Implementierungsdetails haben mir sehr geholfen, die Qualität dieser Arbeit zu steigern.

Ich möchte mich außerdem bei allen bedanken die mich während dieser Arbeit unterstützt haben, vor allem für das Korrekturlesen und die wertvollen Gespräche, die mich zum Nachdenken angeregt haben.

Acknowledgements

I would like to thank my advisors Thomas Preindl and Patrick Denzler for their guidance and consistent support throughout my thesis. They provided helpful insights and feedback especially during the research phase of this work, which aided me in putting my ideas in concrete terms. The periodic discussions about fault-tolerance in stateful applications, different overall approaches and implementation details helped me a lot to raise the quality of this thesis.

I would also like to thank everybody else who supported me throughout this thesis, especially for the proofreading and valuable conversations that set me thinking.

Kurzfassung

Das *Fog Computing* Paradigma ermöglicht die Softwareverteilung von Dienstleistungen in unmittelbarer Nähe zu den Endbenutzer:innen, bzw. in den vorherrschenden heterogenen Umgebungen am Rand des Netzwerkes. Kubernetes verwaltet die Zuteilung und orchestriert die Kommunikation von container-basierten Applikationen, da sie lediglich einen isolierten Ausführungskontext bereitstellen. Es stellt eine benutzerfreundliche Plattform vor allem für *zustandslose* Services bereit, da sie keine zusätzliche Konfiguration bezüglich fehlertoleranter Methoden brauchen. *Zustandsbehaftete* Anwendungen erfordern allerdings eine Umsetzung solcher Methoden, da ansonsten ihr *interner Zustand* verloren gehen könnte, falls sie aufgrund eines Versagens abstürzen.

Diese Bachelorarbeit befasst sich daher mit zwei fehlertoleranten Methoden, die mit ihren unterschiedlichen zugrunde liegenden Architekturen auf einem Kubernetes Cluster bereitgestellt werden. Die *zentralisierte* Methode verfolgt den traditionellen Ansatz, der sich um die Speicherung von Backups an einem einzigen Ort dreht. Die *dezentralisierte*, fehlertolerante Methode setzt state-machine replication (SMR) um, indem ein Server repliziert wird und alle eingehenden Anfragen von Benutzer:innen auf all diesen Instanzen in der gleichen Reihenfolge ausgeführt werden. Dabei durchläuft jeder Server die gleichen Zustandsänderungen und erzielt anschließend identische Resultate. Die Bibliothek *Hazelcast* wird dafür verwendet, da sie den Konsensus-Algorithmus Raft implementiert, der es ermöglicht, konsistente Daten auf mehreren Servern zu verteilen. Beide fehlertoleranten Methoden werden als Container auf einem Kubernetes Cluster bereitgestellt. Darüber hinaus wird jede Methode in eine zustandsbehaftete Anwendung integriert, die mit ihrem flüchtigen Zustand reale Interaktionen von Benutzer:innen simulieren soll, die auf einen Server zugreifen und die darauf gespeicherten Daten verändern.

Die Evaluierung der zustandsbehafteten Applikation und der beiden fehlertoleranten Methoden zeigt, obwohl der dezentralisierte Ansatz mehr Zeit braucht, um den Betrieb nach einem Absturz wieder aufzunehmen, übertrifft er letztendlich den zentralisierten Ansatz in Bezug auf der Speicherung und des Abrufens von Zuständen. Die zentralisierte fehlertolerante Methode schneidet allerdings gleich nach dem Start besser ab und benötigt weniger Speicher während des ganzen Evaluierungszeitraums. Obwohl beide Ansätze mit einer unterschiedlicher Anzahl an Kopien eingesetzt werden, erreichen sie scheinbar konstante Resultate.

Abstract

The *fog computing* paradigm enables the deployment of services near end-users, i.e., in predominant heterogeneous environments at the edge of the network. Kubernetes manages the deployment and orchestrates the communication of *containerised* applications since they merely provide isolated execution contexts. In particular, it provides an easy-to-use platform to deploy *stateless* services since they do not need additional configuration concerning fault-tolerant methods. However, *stateful applications* do require implementing such methods, as their *internal state* can be lost in case they encounter a failure and crash.

Therefore, this bachelor thesis presents two fault-tolerant methods with different underlying architectures deployed on a Kubernetes cluster. The *centralised* method follows the traditional approach of storing backups at a single location. The *decentralised* fault-tolerant method implements state-machine replication (SMR) by replicating servers and executing incoming client requests in the same order on all those instances. In doing so, each server follows equal state changes and produces identical results. The library *Hazelcast* is used to consistently distribute data onto multiple servers as it implements the consensus algorithm Raft. Both of the fault-tolerant methods are containerised and deployed on a Kubernetes cluster. Moreover, each is integrated within a stateful application with an ephemeral state that simulates real-world interactions of clients accessing stateful servers and modifying stored data.

The evaluation of the stateful application and each fault-tolerant method shows that although the decentralised approach takes more time to resume its operation after it crashed, it eventually outperforms the centralised approach in terms of the persisting and retrieving states. However, the centralised fault-tolerant method performs better right after startup and generally requires less memory during the whole evaluation period. Despite both approaches being deployed with a varying number of replica, they seemingly provide constant results.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The advent of *fog computing* allows the deployment of services at the edge of the network, thus positioning them in closer proximity to end devices, as opposed to the more traditional *cloud computing* paradigm [1]. *Virtualization* techniques enable the allocation of components with isolated execution contexts in heterogeneous environments [2] and, as a result, are the preferred method for deploying services on the fog platform. *Containerization*[1] provides a lightweight approach for bundling the code and all needed dependencies of virtualised applications into a single image. Nonetheless, the isolated nature of container-based services establishes the need for a coordination framework that manages the interplay of the above mentioned applications.

Kubernetes[2] automates the deployment and maintenance of containerised applications. It provides an array of different components to facilitate service discovery and coordinate the replication process of deployments. Kubernetes monitors the health of containers and redeploys the ones that stop functioning, hence improving the *availability* of the deployed service. Therefore, a set of replicated instances of the same container is operating by its specification. *Stateless* replica act as interchangeable instances when deployed, whereas the replication of *stateful* services results in distinctly identifiable instances that require additional configuration[3]. In detail, external mechanisms must be considered to restore the unique state of a stateful replication after it is redeployed. *Fault-tolerant* methods with varying underlying approaches can be integrated with stateful applications deployed within Kubernetes.

State-machine replication (SMR) is a fundamental approach for providing fault-tolerant services in distributed environments [3]. The general concept is based on the idea of

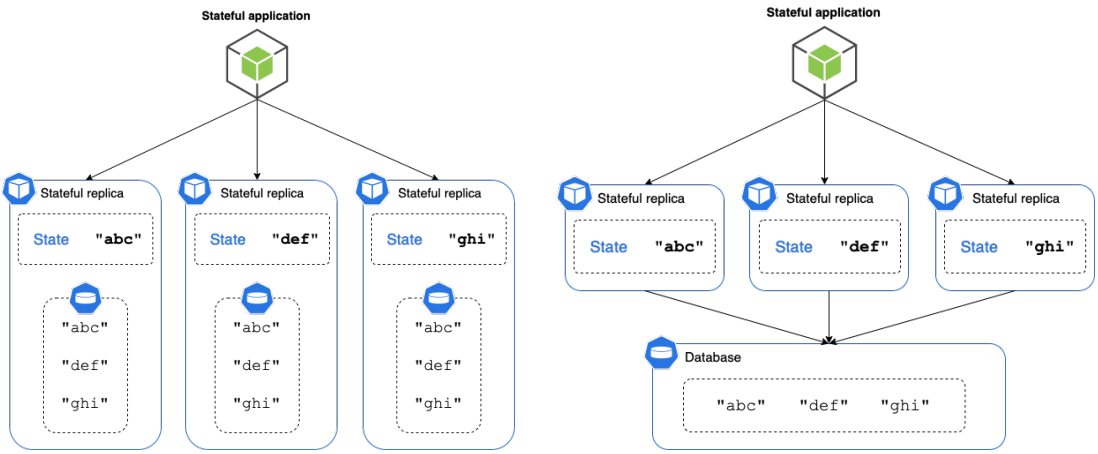
[1] https://www.docker.com/resources/what-container
[2] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[3] https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

executing client requests on replicated server instances and, in doing so, ensuring that each machine undergoes identical state changes. It is essential to the consistency of the stored states that all servers execute incoming requests in the same order. Therefore, the overall *fault-tolerance* is increased, and the distributed service can remain operational even if some replica fail. *Consensus algorithms* constitute the foundation of SMR, since they facilitate the distribution of consistent data within the system [3]. The implementation of *leader-based* replication approaches defines a common hierarchy for servers of a distributed system, for instance, by the *Raft algorithm* [4].

Apart from the *decentralised* approaches mentioned above, *centralised* solutions can also be implemented to increase the fault-tolerance of stateful services, e.g., with backups stored on a separate machine. Kubernetes does not provide an out-of-the-box integration of any of the methods above; however, it allocates the needed tools to deploy those fault-tolerant methods and monitor their interplay with stateful applications.

1.2 Aim of the Work

This bachelor thesis aims to implement two fault-tolerant methods with different underlying architectures and compare their interaction with a stateful application, as depicted in Figure 1.1. The *decentralized* solution provides an implementation of SMR, whereas the *centralized* method presents the more traditional approach of storing backups at a single location. The stateful application is implemented in *JavaScript* and replicated to provide a set of instances that can act independently from each other.



(a) Decentralized (b) Centralized

Figure 1.1: Fault-tolerant methods following different approaches

Regardless of the integrated method, each *stateful replica* generates a unique state that is updated within a fixed time interval. This behaviour emulates state changes performed by clients in real-world scenarios. The decentralised approach distributes the state of each application instance to the other replicas, whereas the centralised approach persists all states in an external database.

The stateful application and both fault-tolerant methods are deployed on a Kubernetes cluster, whereas their interaction is evaluated in terms of performance and scalability by examining the metrics collected by the *Prometheus*[4] service. These metrics include the startup delay, the memory usage and the latency of persisting and retrieving states. It is worthwhile mentioning that the Kubernetes cluster and the evaluation framework only needed minor initial configuration and were already set up.

1.3 Methodology

The literature review conducted at the start of this thesis covers essential concepts and terminologies that are relevant in the context of replicating stateful applications, thus provided the related work and background for this work.

Based on those findings, two different fault-tolerant methods have been chosen and implemented. They both are integrated within a stateful application that maintains an ephemeral state. Each of the implementations has been tested by applying different deployments with a varying number of replica to the Kubernetes cluster. Furthermore, they were exposed to intentionally generated *failures*. Finally, both approaches have been evaluated and compared by the performed tests.

1.4 Structure of the Thesis

The remainder of this thesis is structured as follows: Chapter 2 explains several relevant concepts regarding replicating stateful applications, including different consensus algorithms and characteristics of SMR. Chapter 3 presents an overview of related works induced by the literature review and based on the discussed technologies. The process of integrating both fault-tolerant methods with the stateful application is provided in Chapter 4. The evaluation of the implementations is described in Chapter 5. Chapter 6 concludes this thesis with a summary of the obtained results and remarks on future research.

[4] https://prometheus.io/docs/introduction/overview/

CHAPTER 2

Background

In this chapter, relevant terminologies and concepts in the context of replicating stateful applications are explained. Section 2.1 describes the fundamental procedure of two consensus algorithms and outlines their distinct characteristics. Subsequent to Section 2.2, an overview of leader-based replication, Section 2.3 describes the characteristics of a system implementing SMR. The chapter concludes with Section 2.4, a description of practical byzantine fault-tolerance (PBFT).

2.1 Consensus Algorithms

The predominant protocol *Paxos* [5] affected the development of the *Raft algorithm* [4], which provides an enhanced structure and superior understandability in contrast to the consensus algorithm proposed by *Lamport*. Nonetheless, variations of Paxos are still used to solve the *consensus problem*, as shown in [6, 7, 8]. In general, achieving consensus in distributed environments is at the core of providing fault-tolerant services.

2.1.1 Paxos

The consensus algorithm *Paxos* was originally proposed by Lamport over two decades ago [9]; however, due to its inherent complexity, a more understandable explanation was released three years after its initial publication [5]. The algorithm reaches consensus in a set of *processes* by classifying each participant into one of three groups of agents: *proposer*, *acceptor* or *learner*. The goal of Paxos is to select one value out of multiple choices that are put forth by the processes. Although a single process can allocate all three roles simultaneously [5], the following description uses processes allocating only one type of agent at a time. Paxos solves the consensus problem by splitting the procedure into two phases:

First phase

A proposer sends a proposal, the so-called *prepare request*, with a unique proposal number n to a set of acceptors. It has to be taken into account that the selected quantity of recipients has to comprise a majority of acceptors since each can at most accept a single value, and two majorities out of the same set share at least one common member.

Upon receipt of the prepare request, each acceptor responds to the highest-numbered *proposal* it has already accepted and asserts only to accept prepare requests if they contain a proposal number greater than n. In case n is smaller than the highest-numbered prepare request, the proposal is ignored.

Second phase

After the proposer receives responses from the majority of acceptors, it sends a subsequent proposal, the so-called *accept request*, to the same set of acceptors that responded to the initial prepare request. The accept request contains the highest-numbered response received in the first phase, denoted as v and the same proposal number n. If the acceptors have not responded to the prepare request, the value of v is undefined.

An acceptor only approves the second proposal if it has not already responded to a prepare request with a proposal number greater than n. After the approval of the majority of acceptors, the accepted value is sent to a designated *learner*, and the respective operation is considered as committed. In the final step, the learner distributes the chosen value to the remaining learners.

2.1.2 The Raft Algorithm

The *Raft algorithm* was introduced by *Ongaro et al.* [4] and designed to address shortcomings induced by Paxos, including poor applicability and intricate understandability. Raft facilitates the consensus problem by reducing the state space and by *decomposing* of the algorithm, i.e., the separation into mostly independent subproblems: *leader election*, *log replication* and *safety*. Furthermore, it automates the process of *membership changes*, thus, enabling the participants to rejoin the cluster while ensuring the safety of the algorithm [4].

As mentioned above, consensus algorithms form the foundation of providing fault-tolerant services in distributed environments, for instance, in implementations of SMR [3]. In general, there is a need to consistently maintain a *state* within a cluster of multiple machines. Section 2.3 describes the leader-based process of distributing identical instances of the same data in more detail. The following, however, focuses on explaining the procedure of the Raft algorithm itself.

Basics

Similar to Paxos, each member - or *server* - in the cluster adopts one of three roles at any given time, namely *leader*, *follower* or *candidate*. As a result, the behaviour of each server is determined by the specific tasks involved with each role:

- **Leader:** The single server that takes on this role receives all client requests, replicates log entries, and authorises other servers to commit approved entries.
- **Follower:** This role represents the initial behaviour of each server. A follower responds to inquiries of other servers. However, it generally remains passive in the cluster, for instance, by forwarding client requests to the leader.
- **Candidate:** A follower can become a candidate if a new leader election is initiated. It either changes to its initial role or becomes the new leader once the election is completed.

For the algorithm to function correctly, at least a majority of fully operational servers is required. Raft reaches consensus by working in *terms* of arbitrary length. A leader election initiates a new term with a unique term number. If a new leader is chosen, i.e., the election is successful, the current term is continued. Otherwise, the term number is monotonically increased, and a new election round starts.

Leader election

The election process is initialised by a follower seeking to become the new leader. After the *election timeout* elapsed, it transitions to the role of a *candidate* and broadcasts a *vote request* with the current term number n to the network. A follower rejects the pending vote if it has received prior requests with term numbers greater than n. If a majority of followers accepts the vote, the candidate wins the election and becomes the leader of the current term.

Since followers are unaware of others seeking to become the leader, a vote request can result in a *split vote* if two or more servers simultaneously initiate the election process. In this case, the current term is discontinued, as no candidate can get a majority of responses from the followers. The leader election fails, and a new round starts after the election timeout elapses.

Log replication

The newly elected leader can now receive requests from clients, whereas the committed operations are represented through a *replication log* distributed among all servers in the cluster. Upon receipt of a client request, a *tuple* consisting of the current term number and the sent data is appended to the replication log of the leader. Subsequently, the new entry is replicated from the leader to the followers, which updates their replication logs if the latest entries are consistent with the remaining log of the leader. Otherwise, the

respective follower notifies the leader of outdated log entries to resolve the inconsistency. Therefore, the follower removes all conflicting entries until the first entry shared by both servers is found. In doing so, the replication log remains consistent, and each follower executes the incoming requests in the order specified by the leader. The operations are committed if most followers adopt the new log entry from the leader.

Safety

The Raft cluster implements additional mechanisms to ensure the *safety* of the algorithm for various scenarios. For instance, only candidates with replication logs containing all committed operations are permitted to initiate a new election round. If the leader crashes during replicating log entries to the followers, a future leader attempts to resume the replication. However, the most prominent safety feature of Raft is the *unidirectional* flow of distributing data in the cluster, resulting in improved safety and better understandability of the algorithm.

2.2 Characteristics of Leader-based Replication

Leader-based replication approaches designate a single server to manage the replication and distribution of client requests in the cluster, as shown by the Raft algorithm. The leader obtains all requests that *manipulate* data in the cluster, whereas inquiries to solely retrieve information can be received by any type of server. Since no consensus has to be reached when handling *read-requests*, the *throughput* of the system is increased [10].

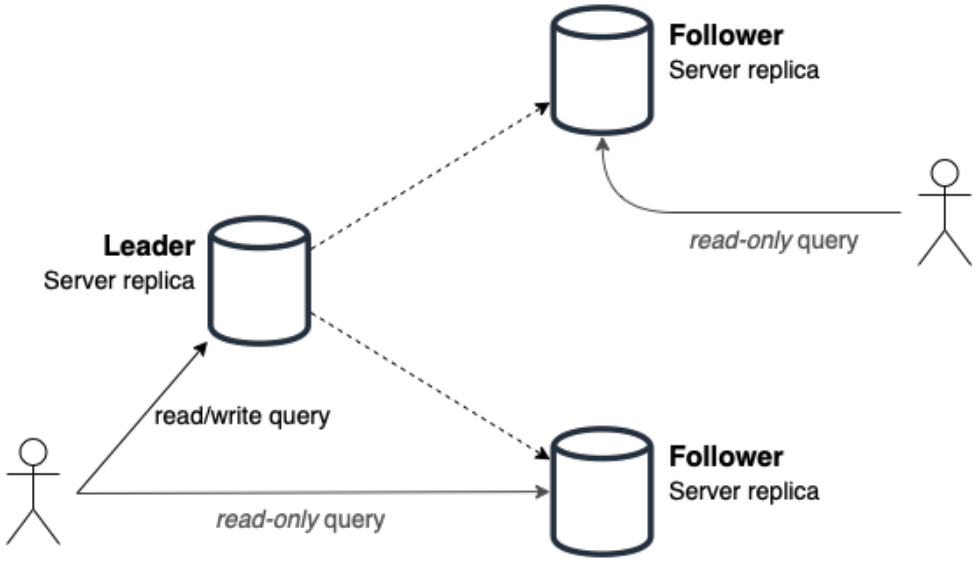


Figure 2.1: Clients accessing servers that follow a leader-based replication approach

Figure 2.1 depicts a cluster with three servers, a leader, and two followers. A client can send a *write-request* that is obtained by the leader, which replicates the changes to the

other servers, thus, storing the data on multiple machines. The cluster continues to operate properly as long as a majority of servers are functioning correctly. The distributed nature of leader-based replication approaches keeps server replica nearby clients, thus, reducing the latency [10].

The *single-leader* approach that is depicted above concentrates incoming write requests to one server. Hence, the *write-throughput* of the cluster is immensely limited. *Multi-leader* approaches enable multiple machines to manipulate stored data; however, the increased complexity usually prevails over the added benefits [10].

2.2.1 Asynchronous and Synchronous Replication

The leader can either wait for the confirmation of a follower that the operation is completed *(synchronous)* or it continues its execution without listening to any responses *(asynchronous)* once it replicates a log entry. The former approach guarantees that the data on the follower's side is consistent; however, the leader cannot process any other client requests during this period since the followers must not lag. As a result, it is impractical to implement synchronous replication [10] solely.

Therefore, asynchronous replication enables the leader to neglect the immediate response of a follower. However, client requests that are in the process of replicating can get lost if the leader encounters a failure and crashes. The *semi-synchronous* model compensates for the drawbacks of both replication approaches by designating all but one follower to operate asynchronously. In case of the synchronous follower crashes, another one steps in to ensure that the data is consistently replicated [10].

2.2.2 Eventual Consistency

As described above, the leader does not wait for the asynchronous follower to respond if the data is successfully replicated. Subsequent *read-requests* can therefore return outdated information since it is not assured that all asynchronous followers store the most recent version of the leader's replication log. However, the followers will get consistent once the pending operations are committed. The inconsistent period cannot be specified precisely. Thus, it can be different for each asynchronous follower. This temporal effect is also known as *eventual consistency* [10].

Strong consistency, as induced by synchronous followers, can be implemented by integrating the concept of *linearizability*, i.e., the outward appearance of one homogenous system with atomic operations, whereas - in the background - it might maintain multiple instances of the same data on different machines [10].

2.2.3 CAP Theorem

The consistency above can further be classified by the CAP theorem [10]. In general, only two of the three involved concerns - consistency, availability, partition tolerance - can be satisfied - while neglecting the residual one. The fault-tolerant methods presented in this

thesis follow different approaches concerning the CAP theorem, as explained in more detail in Chapter 4. A thorough explanation can also be found in the documentations of the used technologies.[1,2]

2.2.4 Crash-Recovery Faults and Byzantine Failures

Leader-based replication approaches need to identify and resolve server failures, besides dealing with the above-described timing assumptions and consistency concerns. For instance, a server stops responding abruptly, whereas the most likely explanation is that it *crashed*. After an unspecified length of time, the server can recover from this *fail-stop behaviour* and resume its intended operation [10]. Although all the data in-memory is assumed to be lost, prior state information can be retrieved from a *stable storage*. Raft generally supports this procedure, provided that the cluster at least consists of $2f + 1$ participants, whereas f, denotes the number of faulty servers. However, a server may not recover from a crash and thus stop responding [10].

Apart from the fail-stop behaviour explained above, a server can also show an *arbitrary*, *byzantine*, behaviour. As a result, it responds with wrong messages or may even attempt to deceive other servers in the cluster maliciously. A wide range of consensus algorithms, including Raft, operates in *non-byzantine* environments; however, several algorithms are solving the Byzantine consensus problem [11].

2.3 State-Machine Replication

SMR is a fundamental approach for providing fault-tolerant services in distributed environments [3]. A system implementing SMR remains operational even if some of the servers fail, since data is replicated and distributed onto multiple machines. Furthermore, it designates a *strong leader* to accept client requests, thus, adheres to the leader-based replication approach discussed earlier. A state-machine maintains its state in form of *state variables* that can be manipulated by applying a set of *commands* [3]. In general, state-machines implement two commands in order to provide basic functionality, i.e., allowing clients to perform *read* and *write* operations on the state variables. Modern systems realize those commands in form of *functions* that *deterministically* produce an output to a provided input.

Therefore, the output of each function only relies on the input and the current state of the corresponding replica. Since all servers in the cluster execute client requests in the order proposed by the leader, each one progresses through the same state changes and subsequently generates identical results [3]. SMR implements the concept above of linearizability, thus appearing to clients as a homogenous system while hiding the underlying complexity of managing multiple machines.

[1] https://docs.hazelcast.com/hazelcast/5.0/consistency-and-replication/consistency
[2] https://docs.couchdb.org/en/stable/intro/consistency.html

2.3.1 Optimizing for Scalability

SMR allows servers to recover from the fail-stop behaviour described above. Hence, the availability is only determined by the number of replica and can, therefore, effortlessly be improved by adding further ones. However, the *scalability* and *throughput* remain limited, since each replica has to execute all incoming client requests [3]. *Partial replication* attempts to solve these restrictions by partitioning the state onto multiple replica, as shown in [6, 7, 12, 13].

2.4 Practical Byzantine Fault Tolerance

PBFT was initially introduced by *Castro et al.* [14] and is considered as the first solution to the byzantine consensus problem in the *partially-synchronous* model. The algorithm is implemented on the basis of state-machine replication, thus resulting in similar procedures. For instance, the *backups* execute client requests in the order proposed by the *primary*. However, PBFT requests that at least $3f + 1$ servers are participating in the cluster, whereas f denotes the maximum number of faulty replica. Besides the *primary-backup* hierarchy, PBFT integrates *quorum replication* in order to compensate arbitrary behaviour of servers [14]. The following summarizes the procedure of PBFT:

Primary-Backup Approach

As briefly mentioned, the primary-backup approach shares specific characteristics with the leader-based replication model. The procedure is recapped to adhere to the proper terminology. It operates in terms referred to as *views*. A server is designated as the *primary*, which has the authority to accept client requests and propose an execution order for the other servers, the *backups*. A view transitions to its next iteration in case the primary encounters a failure, and as a result, a new primary has to be chosen.

Quorum Replication

A *quorum* represents the minimum number of replicas required to carry out operations in distributed environments and has to adhere to two conditions to function correctly: Firstly, the intersection of two quorums shares at least one common replica that is healthy. Secondly, a quorum exists without any faulty replicas at any given time. Furthermore, a quorum must contain at least $2f+1$ replicas for a request to be reliably stored. However, if $f+1$ servers have replicated the respective request, it exists at least on two healthy replicas.

Standard Procedure

The algorithm operates in three phases: the *pre-prepare*, *prepare* and *commit* phase. Upon receipt of a client request, the primary of the active view distributes it to all backups. They generally send their results back to the client once they receive and execute the request.

In the pre-prepare phase, the primary proposes a sequence to execute the requests and broadcasts it in the cluster. Each recipient appends the requests to its replication log once it verifies the proposal's validity. Each backup informs all other backups about the initial proposal by broadcasting a prepare message and subsequently transitions into the *prepare phase*. After a quorum of $2f$ prepare messages is received, the backup enters the commit phase by broadcasting a *commit message*. The operation is considered executed once it obtains a quorum of $2f + 1$ commit messages. Based on $f + 1$ replies, at least one replica has responded with the correct result, and the client can be sure about its correctness.

Checkpoints and View Changes

Once the three-phase protocol is executed, a replica captures the current state and creates a *checkpoint*. It notifies other cluster members about the gathered information and awaits their approval to store the checkpoint in its replication log. A replica can discard the current state and roll back to a captured state in case $2f + 1$ checkpoint messages are received.

As mentioned above, a primary is chosen if the prior one encountered a failure. A quorum of $2f + 1$ approvals of backups is required to initiate the so-called *view-change*. In doing so, the system can compensate for crashed or arbitrary behaviour of primaries.

2.4.1 Improved Performance with HotStuff

Similar to PBFT, *HotStu* follows a leader-based replication approach and operates in partially-synchronous, byzantine environments. It also utilises a three-phase protocol. However, it immensely reduces communication complexity. The process of correctly selecting a new primary has $O(n^2)$ complexity in PBFT; however, it only requires a single round trip in HotStuff, thus $O(n)$. Furthermore, a view-change in PBFT has an upper bound of $O(n^3)$, while HotStuff achieves it with linear complexity [15].

CHAPTER 3

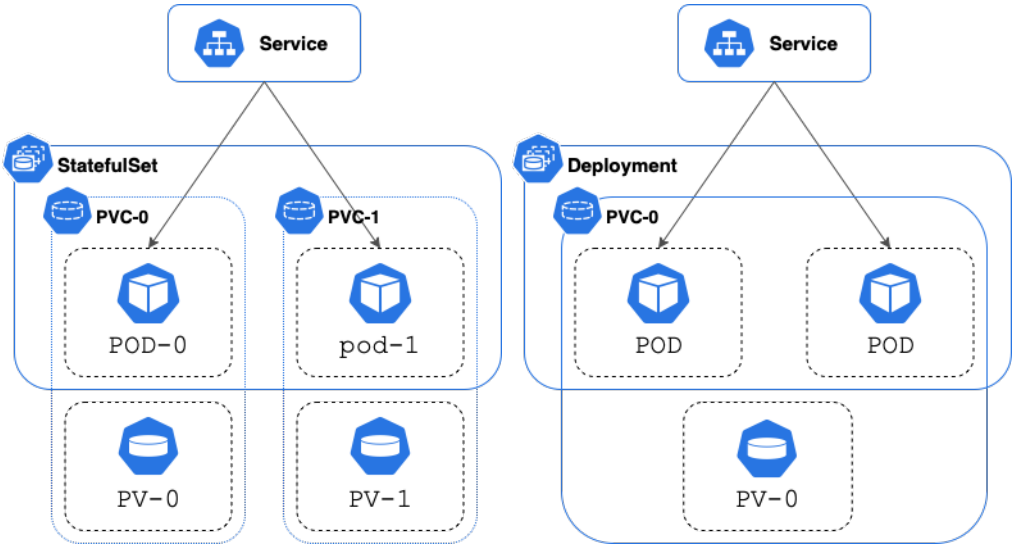
Related Work

The following chapter provides an overview of related works introduced by the literature review. Section 3.1 gives a brief introduction to the built-in components of Kubernetes. In Section 3.2, an implementation of an additional controller is proposed to resolve the limitations of those components. Section 3.3 concludes this chapter with a state-machine replication approach that utilizes shared-memory to distribute requests.

3.1 Built-in Components of Kubernetes

To better understand the findings presented in subsequent sections, the following gives a brief description of the built-in components of Kubernetes. There are several Kubernetes components that maintain the deployment and replication of applications, including the *Deployment* and *StatefulSet* controllers[1]. Based on a single application image, they both deploy a predefined number of *Pods*[2], the smallest deployable unit in Kubernetes, which allocates one or more containers. Kubernetes monitors a Pod's health and redeploys it if it encounters a failure and crash. Thus, the availability of the deployed service is increased. The StatefulSet controller is intended to manage *stateful* services, while the Deployment component is designed to maintain *stateless* replicas. The *Service*[3] component groups a set of Pods based on a common *label* and provides homogenous network resources for all related components. Kubernetes allocates a Persistent Volume (PV)[4] to separate the data from the Pod by storing it externally. If the application fails, the state can be restored from the PV. A Persistent Volume Claim (PVC) is defined by the controller's specification and requests a PV with certain characteristics.

[1] https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/
[2] https://kubernetes.io/docs/concepts/workloads/pods/
[3] https://kubernetes.io/docs/concepts/services-networking/service/
[4] https://kubernetes.io/docs/concepts/storage/persistent-volumes/



(a) StatefulSet Controller (b) Deployment Controller

Figure 3.1: Persistent Volumes attached to different Kubernetes deployments

In Figure 3.1, a common structure of the above-described components is depicted. In both scenarios, the deployed Pods are grouped by a Service component. A PVC requests a PV for each Pod that is deployed with the StatefulSet controller. For instance, *pod-0* solely accesses and interacts with *pv-0*. Since a Pod remains its identity when it is redeployed, the PVC subsequently claims the same PV for it. As depicted in Figure 3.1b, Pods deployed with the Deployment controller act as interchangeable instances when replicated. Hence, a PVC must be defined independently, resulting in a single PV shared by all Pods.

3.2 Issues regarding Availability

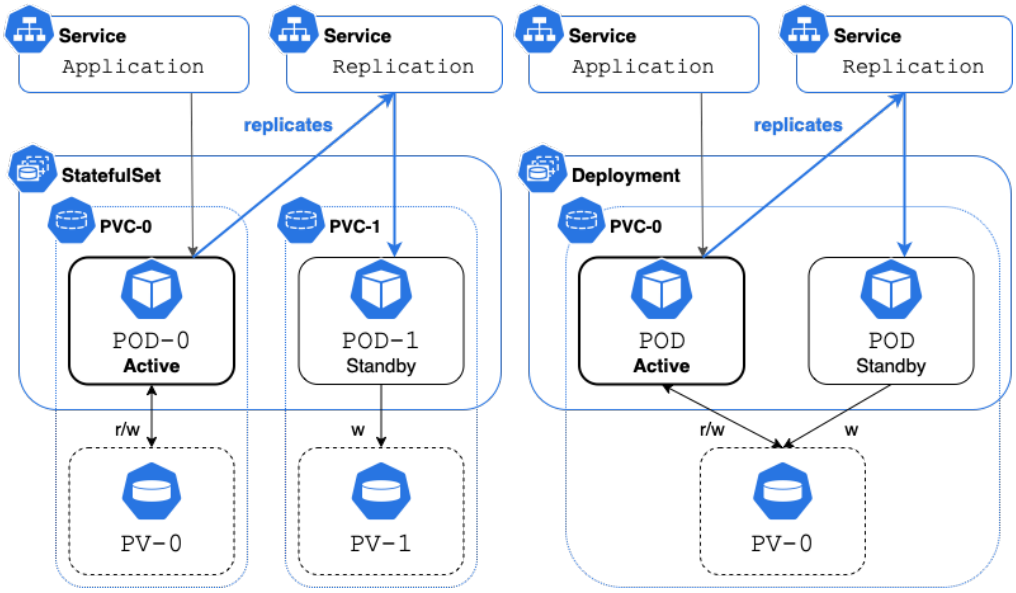
The authors *Abdollahi et al.* [16] identify shortcomings of built-in components of Kubernetes. The built-in controllers of Kubernetes involve two main issues concerning the *availability* of deployed services. Firstly, a Pod that encountered a failure is not responsive until it remedied the error, thus suspending clients from interacting with the deployed service. Therefore, the time to redeploy a Pod, including the time to access the PV, may exceed the time assumptions of highly available services.

Secondly, a StatefulSet controller cannot differentiate between a *node failure* and a *network partition* [16]. Thus, a Pod deployed on an affected node is not responding until the failure is resolved or the network is repartitioned. A StatefulSet controller does not immediately recreate an unavailable Pod but waits until the node is reachable again. It is possible to initiate the redeployment process manually; however, the data stored in the PV will be lost.

The Deployment controller does not consider any data loss; hence, unavailable Pods can be redeployed [16]. However, the location of the stored data that belongs to the crashed Pod cannot be determined since all Pods share the access to a single PV. Furthermore, Pods are generally not aware of failures regarding other Pods. Thus, the data will also be lost.

3.2.1 Integration of Additional Controller

Abdollahi et al. [16] propose the integration of an additional controller, the State controller (SC), in order to fix the previously described issues of the built-in components. The utilization of the SC allows the deployment of highly available stateful applications. It is directly integrated within the Kubernetes environment and deploys two service components, as depicted in Figure 3.2a and 3.2b respectively.



(a) StatefulSet Controller (b) Deployment Controller

Figure 3.2: Deploying stateful applications with additional State Controller

In both scenarios, a secondary label, either *active* or *standby*, is assigned to each Pod that is deployed. In doing so, the relevant Pods are grouped into one of the two aforementioned service controllers. If a Pod has the active label, it is exposed to the network by the *application* service and, therefore, capable of accepting client requests. If a Pod has the standby label, it is grouped by the *replication* service to receive the replicated data from *active* Pods. The label of a Pod is provided through the use of an environment variable.

The SC updates the label of an *active* Pod to *standby*, if it encountered a failure and, for instance, crashed. Subsequently, one of the Pods remained passive until this point became active. If a StatefulSet controller is used to deploy the Pod in question, the promoted Pod can access the PV of the previous Pod. If the Deployment controller is

used, it simply retrieves the state from the same database it accessed with its former identity. Moreover, the SC repeatedly checks if the *entrypoint service*, i.e., the component that maintains the deployment, is still reachable to detect and prevent network partitions. If it is not responding, the SC performs a self-cleanup procedure, terminating all running components in the process.

The authors *Abdollahi et al.* evaluate the integration of their proposed controller by deploying a Video-on-Demand (VoD) service [16]. They perform measurements to compare the SC with the default behaviour of Kubernetes, i.e., implementations using only the built-in components. In detail, the two deployments depicted in Figure 3.2 are applied. Each time a client requests a video, the Pod that received the inquiry persists in its state, i.e., the current playback position, to the attached PV. During their experiments, different components fail intentionally to evaluate the availability's effect. The authors show that their proposed solution enables Kubernetes to compensate for service outages caused by node failures. Furthermore, the integration of the SC with the StatefulSet and Deployment controller drastically improves service recovery after a component encounters a failure.

3.3 Utilization of Shared-Memory

The authors *Netto et al.* [17] integrate state-machine replication directly into Kubernetes. They believe that distributing client requests by utilising *shared-memory* instead of broadcasting them with conventional messages reduces the inherent complexity and provides a more streamlined design. Therefore, the deployed components interact with the key-value store *etcd*[5] to distribute incoming client requests, whereas a custom algorithm coordinates their distribution. The storage runs on a single node. However, it can also be replicated with the Raft algorithm to increase the fault tolerance of the system further.

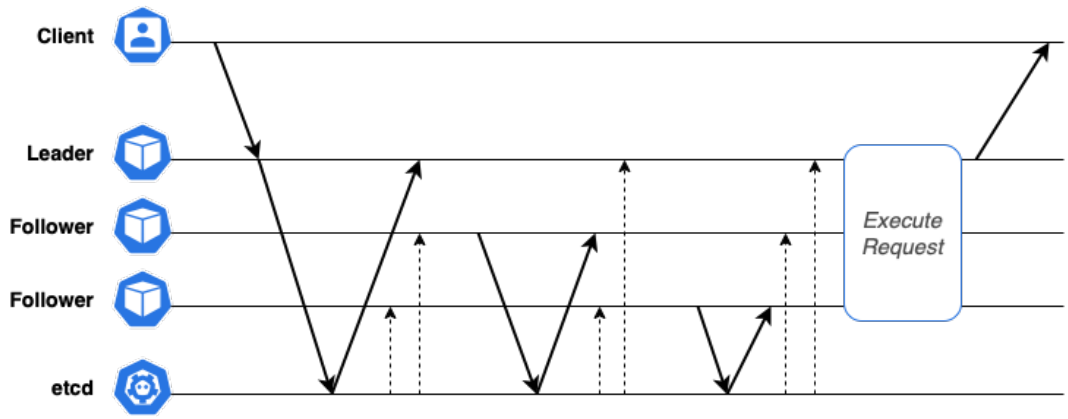


Figure 3.3: Replicating client requests with shared-memory

[5] https://etcd.io

The so-called *DORADO* (**D**ering **O**ve**R** sh**A**re**D** mem**O**ry) protocol coordinates the distribution and replication of client requests by means of the etcd store [17]. Although each replica can receive requests from clients and fetch and persist them in the shared memory, only the recipient is allowed to respond to the initial sender. The communication between Pods and etcd storage is asynchronous, whereas the interaction between clients and Pods is partially synchronous.

In Figure 3.3, the procedure of replicating and distributing a request sent by a client is shown. The recipient, the *Raft leader*, defines an order to the request and stores the corresponding proposal in the shared memory. After it is successfully stored, the other replicas are notified, which will adhere to the order proposed by the leader. They subsequently inform other replicas about their approval by interacting with the etcd store. If the leader collects a *majority* of successful responses, it executes the request and considers the operation as committed. Hence, all replicas executed the request and store an instance of the changed data. The leader then sends the result to the client. If a follower receives the initial request from the client, it is stored in the shared memory, and only the leader is notified. The protocol then continues with the same procedure that is described above.

The authors *Netto et al.* performed several experiments to verify the effectiveness of their proposed solution on the availability and throughput of the deployed service [17]. Furthermore, they encapsulated the DORADO coordination algorithm into a separate component to reduce the application containers' size. They deployed the etcd store only on one node, thus, requiring more communication rounds than a distributed version of the etcd store, since Raft makes it possible to collect multiple messages and send them as one *batch*. During the experiments, the number of clients and replicas varied while the size of the payload of a request remained the same. They conclude that the latency increases while the throughput decreases if more replicas are added to the system.

CHAPTER 4

Implementation

The following chapter provides implementation details on the stateful application and both fault-tolerant methods. Furthermore, different Kubernetes components that are used in the context of this thesis are discussed. Section 4.1 outlines the process of deploying virtualised services within a Kubernetes cluster. In Section 4.2, the intended procedure and structure of the stateful application are described. The chapter concludes with the integration of each fault-tolerant method within the application, which is explained in Section 4.3 and 4.4 respectively.

4.1 Deployment of Container-based Applications

Kubernetes facilitates the maintenance of container-based applications by automating the process of deploying and replicating services[1]. In the context of this thesis, it enables effortless integration of different fault-tolerant methods within a stateful application. Deployments are applied through the use of *containerization*, which is the preferred option over *virtual machines* since the former constitutes the more lightweight approach[2].

4.1.1 Cluster Architecture

Figure 4.1 depicts the structure of the Kubernetes *cluster* that functions as a host of the deployments. It consists of several physical machines referred to as *nodes*. In general, the cluster is organised by two categories of nodes. The single *master* node solely operates control processes, whereas the *worker* nodes carry out application deployments of users. Kubernetes groups its control structures into a separate component - the *control plane* - which is explained in detail in the official documentation[3].

[1] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[2] https://www.docker.com/resources/what-container
[3] https://kubernetes.io/docs/concepts/overview/components/

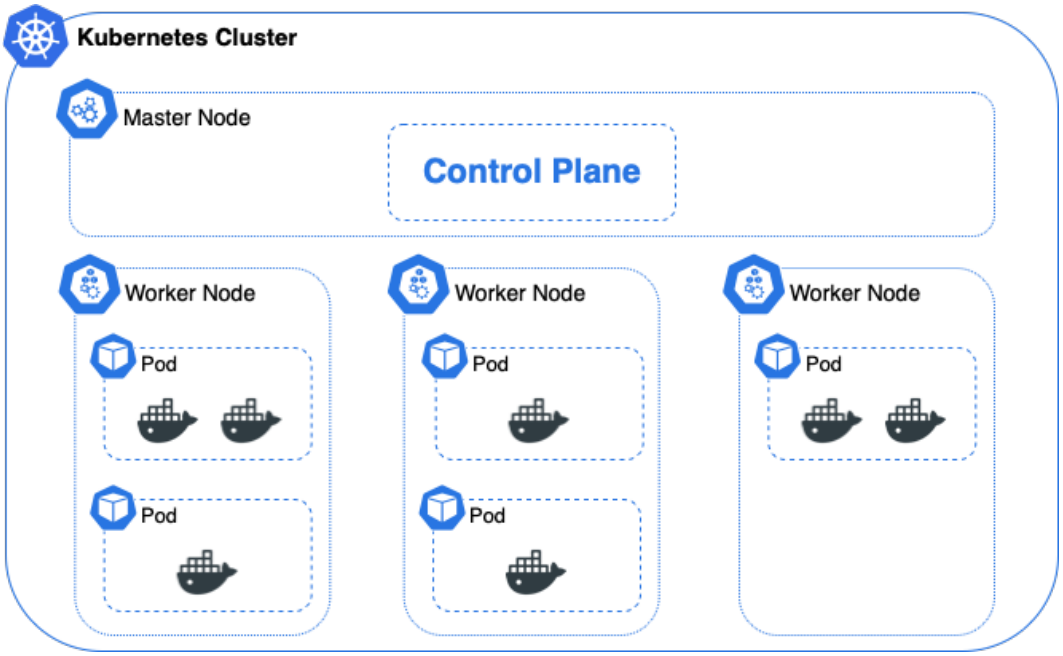


Figure 4.1: Components of Kubernetes Cluster

As already mentioned in Section 3.1, Kubernetes provides homogenous network resources for one or more containers by deploying *Pods*[4]. The controller node maintains the replication of a Pod by instantiating *controller* components, which monitor the health of related instances. If any of them encounters a failure, the respective Pod is redeployed to meet the number of replicas provided by the specification. The cluster is running version *v1.21.4* of Kubernetes. Deployments are verified, maintained and applied through *kubectl*[5]. *Docker*[6] is used to containerize the application and the VPN service *Tailscale*[7] provides access to the cluster from outside the network.

4.1.2 Controller and Service Components

Kubernetes provides an array of different options for maintaining the replication of Pods, including the *Deployment* and *StatefulSet* component. The availability of the deployed service is subsequently increased since both controllers generally attempt to redeploy crashed Pods. However, the Deployment controller is not considered further since its intended purpose is to maintain *stateless* applications. The StatefulSet controller, on the other hand, manages the maintenance of a set of replicated, *stateful*, Pods[8].

[4] https://kubernetes.io/docs/concepts/workloads/pods/
[5] https://kubernetes.io/docs/reference/kubectl/overview/
[6] https://docs.docker.com
[7] https://tailscale.com/kb/1151/what-is-tailscale/
[8] https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

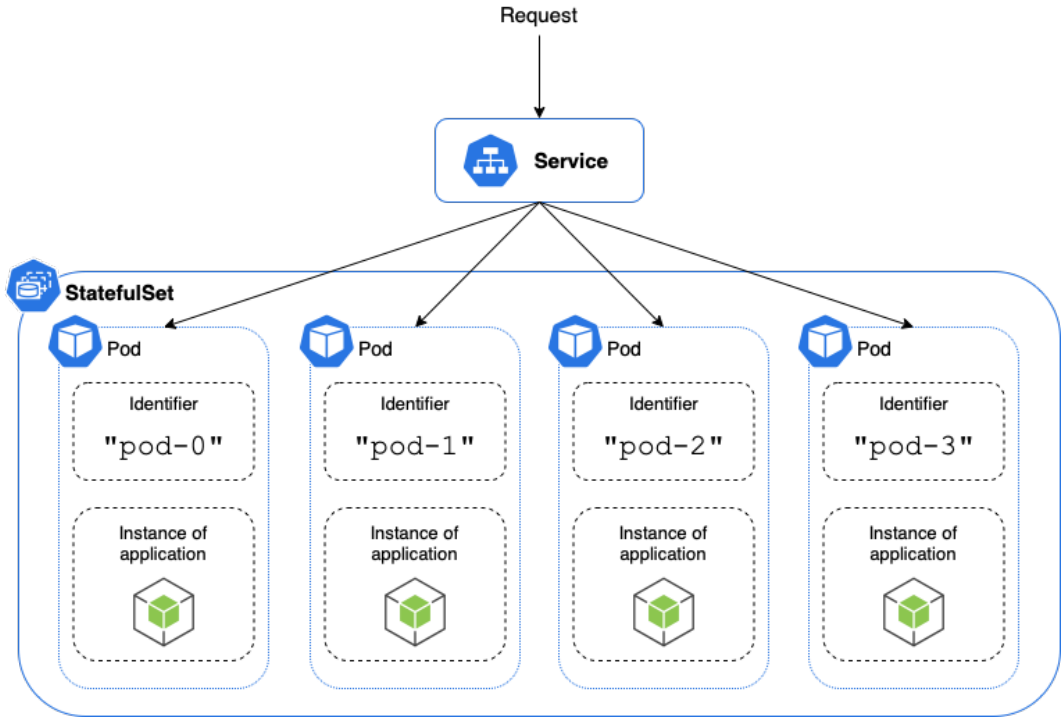


Figure 4.2: Replicated application deployed with Service and StatefulSet components

The StatefulSet controller deploys multiple instances of the specified application image, thus resulting in a set of replicated Pods, as seen in Figure 4.2. Each Pod has a unique identifier and functions as an autonomous application instance. The identity of each Pod remains the same even if it is redeployed; however, the IP address of each replicated instance is dynamic and changes with each rollout.

Therefore, Kubernetes uses an additional abstraction to facilitate access to a set of related Pods. Based on a common *label*, the *Service*[9] component groups affiliated Pods to one logical unit to uphold a static outward view that is independent of the contained resources. Hence, the Service component keeps the same IP address and hostname even if any individual Pods are redeployed. Moreover, it exposes each Pod to the network to be accessed individually while preserving the same URI. Following *http://pod_identifier.service_name*, each replicated Pod can receive requests from clients and other components within Kubernetes.

[9] https://kubernetes.io/docs/concepts/services-networking/service/

4.2 Stateful Application

The stateful application is implemented in *JavaScript* and publicly available on this repository[10]. The main purpose of this application is to equate clients accessing and modifying data of stateful applications in real-world scenarios. However, the application is not required to provide any other function than to maintain an internal *state* and its corresponding updates, as the primary focus of this thesis is the development and integration of fault-tolerant methods. The application uses version *16.14-alpine* of *Node.js*[11] and utilizes *npm* as the package manager. It is designed to be as lightweight and modular as possible.

4.2.1 Generating the Internal State

As described above, the StatefulSet controller deploys a set of replicas, each with a unique identity. Kubernetes maintains these identities by assigning the name specified by the StatefulSet in combination with ordinal numbers. Figure 4.3 depicts Pods with identifiers ranging from *pod-0* up to *pod-[n-1]*, whereas n denotes the number of replicas. The identifier is provided to each Pod by an environment variable. Henceforth, *stateful replica* references a deployed application instance.

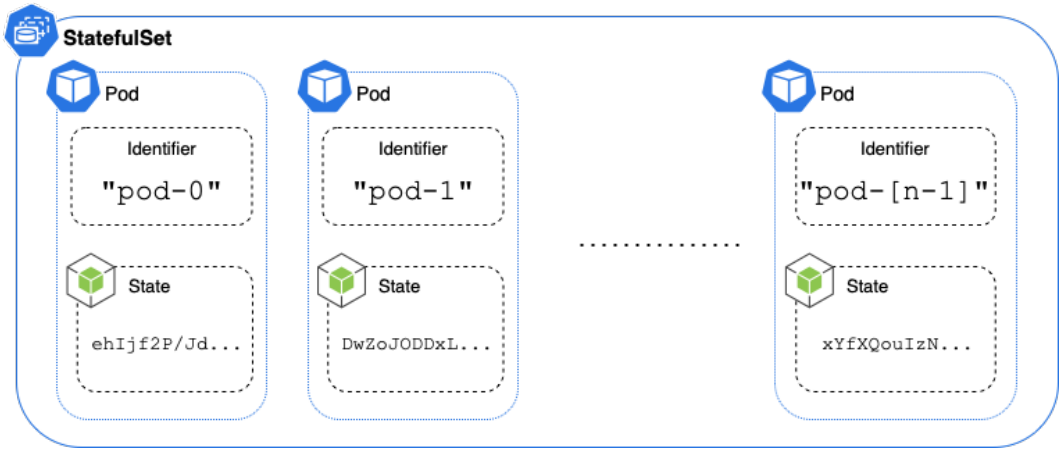


Figure 4.3: Set of application instances with unique identifiers and states

Each *stateful replica* uses its identity as the starting point to generate its internal state, which is the result of a *hash* function. A string with a fixed length represents the state. In detail, a *stateful replica* uses the *SHA256* algorithm to hash the input of the function, and *base64* to encode the result. The state of each replicated instance is valid for a fixed time interval t. Every t seconds the state is replaced by reapplying the hash function to it. Table 4.1 depicts exemplary state changes performed by a *stateful replica* with the identifier *pod-0*.

[10] https://git.auto.tuwien.ac.at/theses/stateful-edge-app
[11] https://nodejs.org/en/docs/

i	State
0	ehIjf2P/Jdj5nd8Oo9CPSw/xvNtpTxwe0t5Y7SWHy2k=
1	h93QgNJEmNejlZXsWKM59TOyLJfWdpedpVLirFow9IU=
2	KQz64JLF67dB/DREluXpB0F4rq3Gpal6kvZD+7cQ1eY=

Table 4.1: State changes performed by *pod-0*

The variable i denotes the i-th state-update and, if multiplied with t, represents the time (in seconds) it takes until the corresponding state is considered valid. Thus, the validity of each state can be expressed with the interval:

$$[i \quad t, (i+1) \quad t)$$

At the beginning of each interval, i.e., after the state is updated, it is persisted with one of the fault-tolerant methods. If a Pod encounters a failure, it restores the last state considered valid. For instance, if the Pod *pod-0* crashes while generating the state for $i = 2$, it restores the state of $i = 1$ since it was persisted with a fault-tolerant method after it was updated. Upon completion of the Pod's redeployment, it retrieves the restored state and continues with its intended operation. The stateful application is kept simple since this thesis focuses on designing and implementing different fault-tolerant methods.

4.2.2 Integration with Storage Adapters

The stateful application integrates different fault-tolerant methods by utilising the key-value store *Keyv*[12] since it supports the implementation of *storage adapters* with custom persisting mechanisms. Thus, the fault-tolerant methods presented in this thesis are implemented as *standalone* storage adapters, whereas each follows a consistent outward appearance. In general, custom storage adapters are required to implement *get* and *set* methods in order to cover basic functionality, i.e., allowing clients to read and write data. Furthermore, it supports TTL based expiry, which can be defined when calling the *set* method. However, the implementations of both fault-tolerant methods do not utilise it since the state already undergoes periodic updates. Additional functionality can be defined by implementing the appropriate functions, e.g., to *delete* certain entries or to *clear* the whole storage.

At the startup of each *stateful replica*, a fault-tolerant method in the form of a storage adapter is provided to the Keyv instance. The environment variable *MODE* instructs it to load either the *central* or *decentral* method. The underlying mechanics of each fault-tolerant method are hidden behind the functions that Keyv provides.

[12] https://github.com/jaredwray/keyv

4.2.3 Procedure of the Application

The application maintains a state by operating in *three steps*. In principle, it initially loads the last valid state, updates and persists it in the database provided by the fault-tolerant method. The steps are executed repeatedly in a fixed time interval. Thus, each *stateful replica* generates new states periodically.

Although the underlying approaches of both fault-tolerant methods are fundamentally different, a uniform application procedure can be defined regardless of the integrated method, as depicted in Figure 4.4. In Sections 4.3 and 4.4, detailed descriptions of both fault-tolerant methods are presented. The application procedure described below applies to the initial startup of a replica and a restart performed due to a failure.

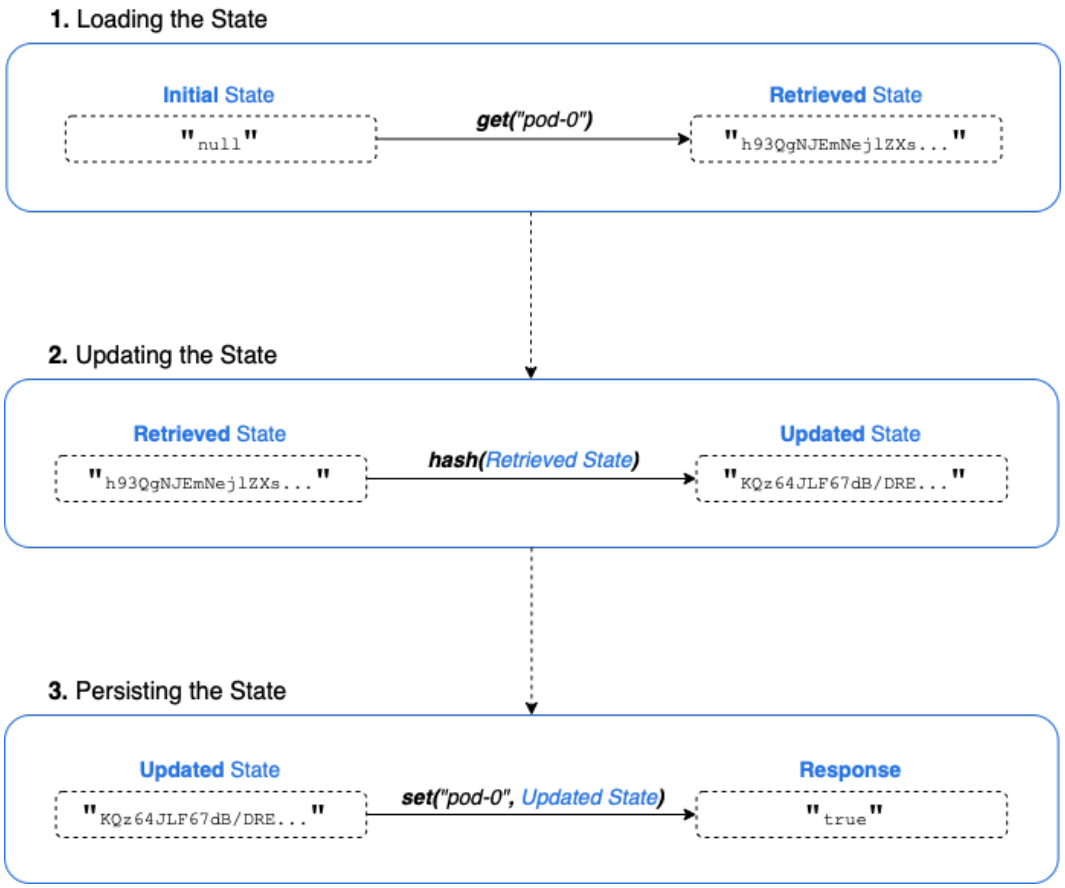


Figure 4.4: Application procedure, independent of the integrated fault-tolerant method

1. Loading the State

At the startup of each stateful replica, it attempts to load the state by calling the *get* function, which returns a promise that resolves to the retrieved value. As described above, the identifier of each instance is used to determine the stored state. If no data is stored

concerning the provided key, the method returns *null*. The actual representation of the persisted states generally differs depending on the integrated fault-tolerant approach.

2. Updating the State

As soon as the stateful replica retrieves the state, it updates it by calling the *hash* method. The built-in *crypto*[13] module is used to create and update states and digest the resulting hash. The method takes the current state as its input and, to generate new states, reapplies the hashing procedure to it. In case the *get* method in the previous step returns *null*, the identifier of each replica is used as the initial input to create the first state.

3. Persisting the State

The result of the hash function is persisted by the integrated fault-tolerant method. The identifier of the stateful replica and the state are provided to the *set* method, which returns a boolean, indicating if the value was stored successfully. Each storage adapter communicates with the fault-tolerant method over HTTP requests. Therefore, the *Axios*[14] client is used.

4.2.4 Application Endpoints

The application uses the *express*[15] package to provide fundamental web functionalities. The procedure described above is embedded into a web server, which allocates two endpoints on port *32000*. In the context of this thesis, they are primarily used for debugging purposes. The following endpoints are reachable for each *stateful replica* within the network of the Kubernetes cluster:

- *GET* / - This endpoint returns a website displaying the identifier and state of the stateful replica.

- *GET* /metrics - This endpoint collects the logged metrics in order for them to be processed by the *Prometheus* service running on the cluster, and subsequently be evaluated.

[13] https://nodejs.org/api/crypto.html
[14] https://axios-http.com
[15] https://expressjs.com/en/4x/api.html

4.3 Centralized Approach

The *centralised* fault-tolerant method follows the traditional approach of storing data at a central location that is independent of the application. Thus the storage is deployed and then runs separately from the stateful application. The deployment of this fault-tolerant method includes a StatefulSet and Service controller that are bundled into and deployed with a *helm chart*[16]. In general, it consists of multiple specifications of Kubernetes components applied at once, thus streamlining the process of using complex deployments.

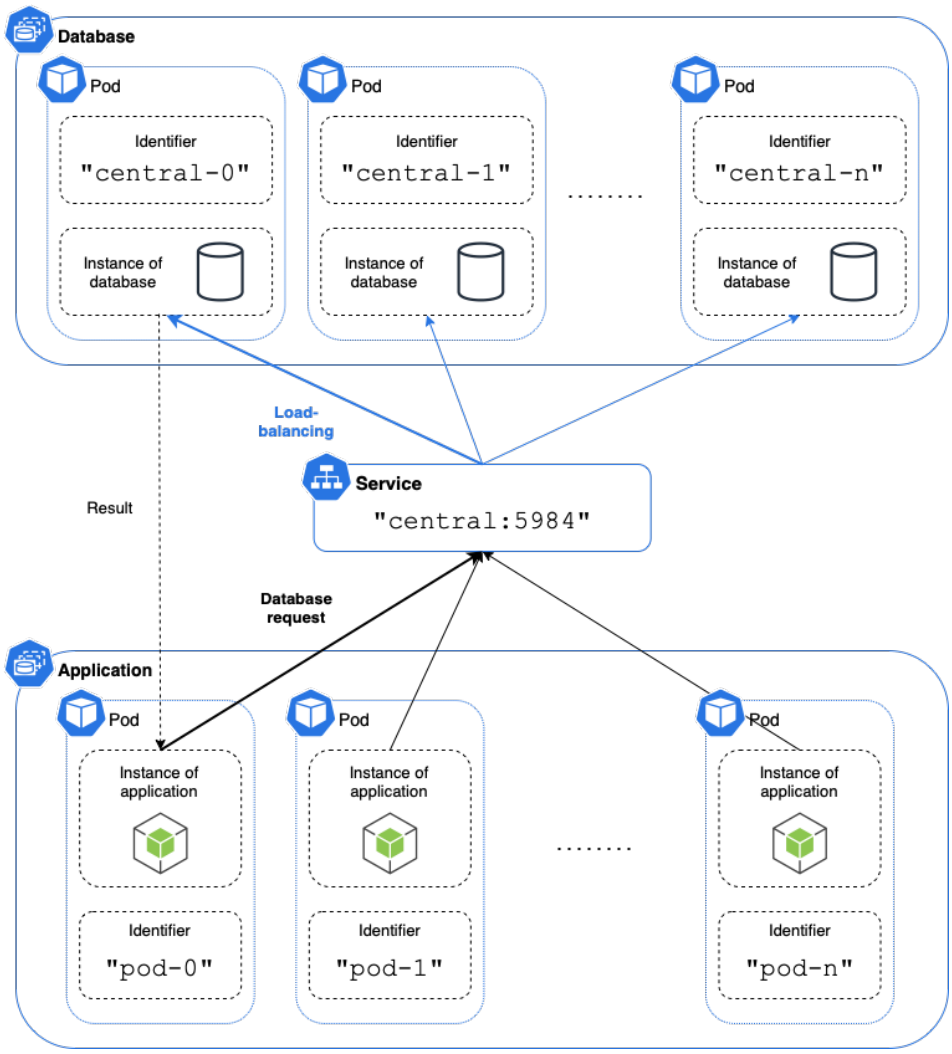


Figure 4.5: Interaction of stateful application and centralized method

In detail, the helm chart comprises a *CouchDB*[17] cluster. Since a StatefulSet controller

[16] https://artifacthub.io/packages/helm/couchdb/couchdb
[17] https://docs.couchdb.org/en/stable/

maintains its deployment, the central storage is replicated, and multiple instances of the same database are created - resulting in a similar set of replicas induced by the stateful application. Although this has no direct impact on the investigations of this thesis, the bottleneck of using a single database is immensely reduced. Data is stored in memory per default. As seen in Figure 4.5, each *central replica* has a unique identifier and holds an instance of the CouchDB database. The IDs are constructed in the same way as the stateful replicas' identifiers, resulting in names of the form: central-i, $0 < i < n - 1$, whereas n depicts the number of replicas. The helm chart deploys three replicas per default.

The set of *stateful replicas* can access the database using the Service controller, which allocates the static IP address *http://central:5984*. In general, incoming requests are automatically load-balanced overall healthy *central replicas*. As depicted in Figure 4.5, the Service component forwards the request of *pod-0* to *central-0*, which processes it and subsequently responds the result back to the initial sender of the inquiry. The states of all *stateful replicas* are stored in a single database and replicated on multiple instances. In Section 4.3.2, the synchronization of data within the set of *central replicas* is explained.

4.3.1 Structure of Database

As a *non-relational* database, CouchDB stores data in form of *JSON* documents[18]. Each document can contain any number of fields and is uniquely identified by a *name*. CouchDB ensures consistency of distributed data within the set of *central replicas* by using the metadata of each document to keep track of its current state. Since the data of all *stateful replicas* are shared by the same database, a single document is used to represent the state of an individual application instance. Table 4.2 highlights the properties required to persist the state of a *stateful replica*.

Property	Type	Description
_id	string	*Unique identifier* of the document
_rev	string	*Revision number*, used to keep track of applied changes
value	string	*State* of the stateful replica
expires	number	*TTL*, shows the validity of state (in seconds)

Table 4.2: Properties of a CouchDB document

The *_id* property generally resembles the identifier of a *stateful replica*, thus not changing after it is initialised once. The *value* property is updated in a fixed time interval since the state is persisted respectively. CouchDB updates the revision number each time the document is changed. Therefore, a request must contain the currently valid revision number for the correct state to be properly updated. After the state is persisted, the response of CouchDB includes the updated *_rev* property. Since the TTL feature of keyv is not used, the *expires* property is not set and therefore remains *null*.

[18] https://docs.couchdb.org/en/3.2.0/intro/index.html

4.3.2 Synchronization of Stored Data

Since the Service controller load balances incoming requests overall healthy replicas, there is a need to synchronise the data stored on multiple replicas. Therefore, CouchDB features *multi-master* synchronisation to ensure that each replicated instance of the database remains consistent[19]. It guarantees that the data will become consistent after an unspecified amount of time, thus following the *eventual consistency* approach. Based on the CAP theorem, CouchDB chooses *availability* and *partition tolerance* in favour of strong *consistency*. The replication protocol of CouchDB implements the Multiversion Concurrency Control (MVCC) model to synchronise data between two peers. It therefore sends HTTP requests to the API endpoints of CouchDB. The following summarises the procedure that is explained in detail in the official documentation:

The replication protocol is described in the context of a *Replicator* service. Like the database maintaining individual documents by their *_id* property, the Replicator keeps track of the replication history by instantiating replication identifiers. Document changes are synchronized from a *source* to a *target*. Moreover, the flow of information is uni-directional. The Replicator retrieves the replication logs from the source and target after creating the replication identifier. The Replicator then continuously listens to any document changes as of the last checkpoint at which both are synchronised. After a batch of changes is read, the Replicator calculates the difference in revision numbers, i.e., identify all document changes present on the source but not on the target. The Replicator subsequently requested the aforementioned documents and transferred them to the target. After the recipient successfully stores all documents, the Replicator updates the source and target to include this checkpoint. The Replicator then starts waiting for new document changes to arrive. Nothing will be synchronised if the revision difference does not put forth any documents.

4.3.3 API Endpoints

CouchDB provides several endpoints for stateful replicas to manipulate stored data and documents. The following API endpoints are used to retrieve and persist states:

- *PUT* /db - *Creates* a database with the given name **db**. The data included in the body of the request is not relevant, thus can be *null*.
- *GET* /db/doc - *Retrieves* the document with the given name *doc* from the database with the name **db**. The response includes the state of the corresponding stateful replica and the revision number.
- *PUT* /db/doc - *Creates* or *updates* the document with the given name **doc**. The body of the request must include the document identifier, the last valid revision number and the updated state of the stateful replica.

[19] https://docs.couchdb.org/en/3.2.0/replication/protocol.html

4.3.4 Interaction with Stateful Application

Figure 4.6 shows the interaction between a replicated instance of the application and the centralised fault-tolerant method. The requests are forwarded by the Service controller to one of the *central replicas*, in this case, *central-0*. The initial request that accesses the database or an individual document creates the respective resources in case they are not present.

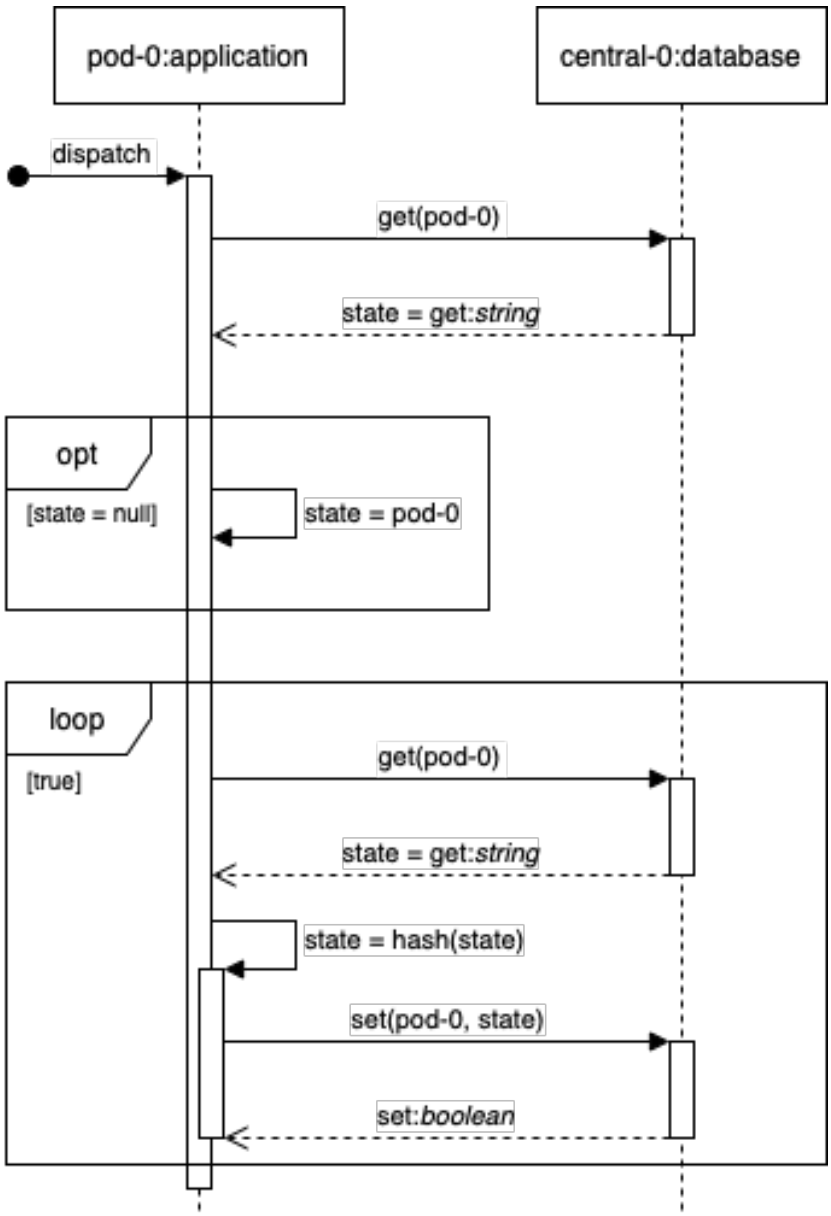


Figure 4.6: Interaction between application and centralized fault-tolerant method

Pod-0 initially attempts to retrieve the state and revision number from *central-0* by providing its identifier to the *get* method. If the response is *null*, the identifier acts as the initial state. The revision number will be used later when the newly updated state is persisted in the database. The retrieved - or initially defined - state is hashed, replacing the old one in the process. The newly updated state is persisted in the database by providing it and the previously retrieved revision number to the *set* method. The identifier *pod-0* is also enclosed to locate the correct document. The database returns the newly generated revision number for the subsequent request to update the stored state. As already mentioned, the steps, i.e., the retrieving, updating and storing procedures, are repeated in a fixed time interval.

4.4 Decentralized Approach

The structure of the *decentralised* fault-tolerant method considerably differentiates from the centralised approach described above. *Bakhshi et al.* [18] show that persistent storage can be implemented at the network's edge. As depicted in Figure 4.7, it is deployed alongside, and at the same time heavily depends on, the stateful application. Therefore, the StatefulSet controller that deploys the application is extended to include the specification of this fault-tolerant method, resulting in the deployment of an additional container. In contrast to the centralised approach, there is no need for a supplementary Service controller to enable the communication between an application and fault-tolerant method since they are contained within one Pod. Henceforth, a deployed instance of the decentralised approach is referred to as *consensus replica*. Consequently, a *stateful replica* communicates solely with its accompanying consensus replica, which distributes the data within the set of all other replicas. Thus, the state of each application instance is stored in multiple locations.

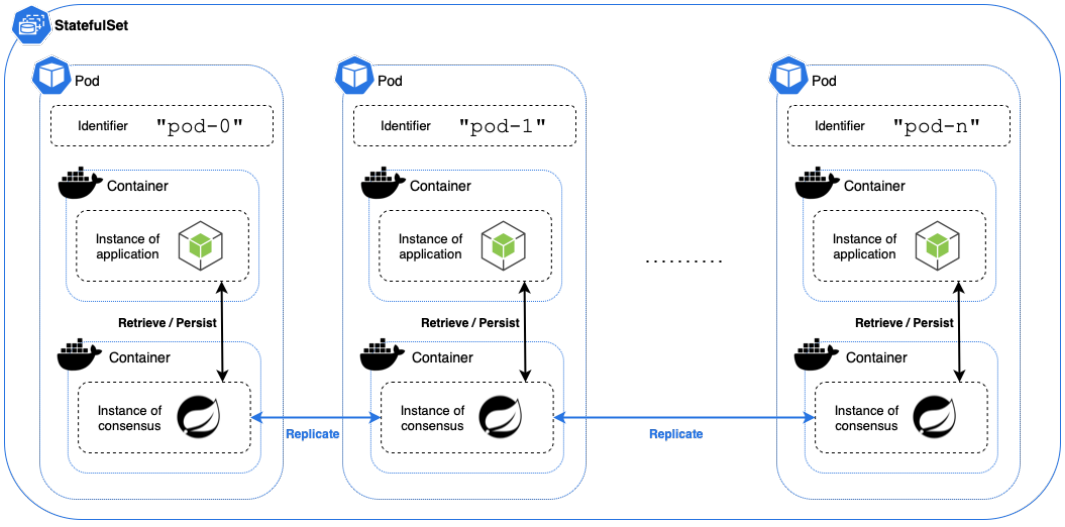


Figure 4.7: Structure of application with integrated decentralized fault-tolerant method

The decentralized fault-tolerant method is implemented in *Java* and features a *Spring Boot*[20] application. The project is built with *Maven*[21] and is available on the same repository alongside the stateful application. The *Hazelcast*[22] library is used on top of the Spring Boot application, as it implements the needed tools to replicate data within the cluster consistently. A Hazelcast instance runs on each consensus replica and is started after the Spring Boot application. The decentralised approach deploys additional Service controllers to enable the automatic *discovery* of other Hazelcast instances. Thus, each consensus replica joins the Hazelcast cluster after it is started.

4.4.1 API Endpoints

The Spring Boot application is mainly used for building a RESTful web service that accepts requests to allow the manipulation of stored data. It provides several endpoints for the stateful replicas to access the functionality of the Hazelcast cluster. Since the primary interaction between the stateful application and fault-tolerant method revolves around retrieving and persisting information, the Spring Boot application provides two corresponding endpoints. Two additional endpoints are used to guarantee the correct behaviour of the CP Subsystem, as described below.

- *GET* /api/v1/state/pod - Retrieves the state for the stateful replica with the name *pod*.
- *PUT* /api/v1/state - Updates stored state. The body of the request must include the identifier and the updated state of the stateful replica.
- *DELETE* /api/v1/cp/pod - Deletes the member with the given name of *pod* from the CP Subsystem component in the Hazelcast cluster, as described in Section 4.4.4.
- *GET* /api/v1/cp/verify - Retrieves information from the cluster to verify if the discovery phase of the Hazelcast cluster is completed.

As already described, the StatefulSet controller of the decentralised fault-tolerant method deploys two containers simultaneously, i.e., the stateful and consensus application. The former awaits the completion of the formation of the Hazelcast cluster since state changes can get lost as a pending Hazelcast member cannot receive any messages. Therefore, the stateful replica constantly checks if the Hazelcast cluster is initialised, i.e., all members joined successfully before it starts with its indented operation.

[20] https://spring.io/projects/spring-boot
[21] https://maven.apache.org
[22] https://docs.hazelcast.com/hazelcast/latest/

4.4.2 CP and AP components

In contrast to the centralized approach, the decentralized fault-tolerant method prefers *consistency* over *availability* - with respect to the CAP theorem. The Hazelcast library provides strongly consistent data structures with its *CP Subsystem*[23] component. However, it needs to be enabled manually since Hazelcast solely supports *AP* components per default. At startup, each consensus replica specifies the size of the CP Subsystem cluster, which is provided by the environment variable *REPLICAS*. Hazelcast forms a cluster of all those members and grants them access to the *CP* components of the library.

4.4.3 Members and Groups

The set of consensus replicas accesses the data structures of the CP Subsystem in the form of smaller *groups* of three to seven members. Each group implements the *Raft algorithm* and elects its leader to achieve consensus. Hazelcast automatically creates the *METADATA* group, which coordinates allocating members in other groups. If not specified otherwise, the members in the CP Subsystem only use data structures belonging to the *DEFAULT* group. As depicted in the exemplary Hazelcast cluster shown in Figure 4.8, two custom groups with three members each are created. Although a member can be part of multiple groups, it solely interacts with the data structures related to one designated group.

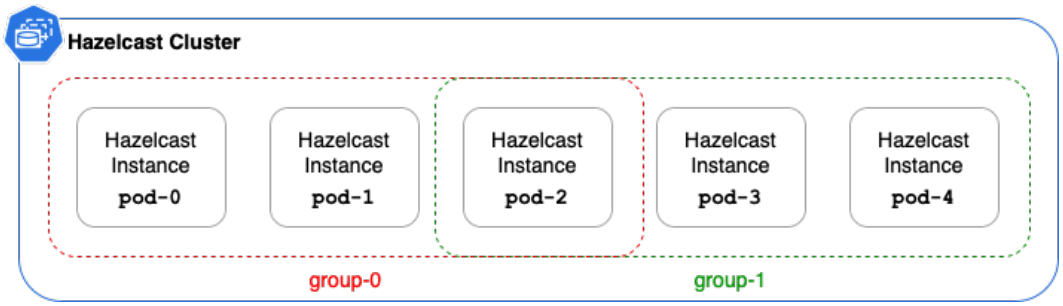


Figure 4.8: Custom CP Subsystem groups sharing a common member

The particular group of a CP member is determined by the ordinal index of the corresponding consensus replica's identifier. With a set of consensus replicas of size n, a group size g, a total number of groups $m = \frac{n}{g} + 1$, j depicts the ordinal index of the group. Thus, for each consensus replica i, it holds true:

$$i \epsilon \{0, ..., n-1\} : j = \frac{i}{g}, 0 \leq j \leq m-1 \tag{4.1}$$

As shown in Figure 4.8, a cluster of size $n = 5$ and a group size of $g = 3$, results in a total number of $m = 2$ groups, namely *group-0* and *group-1*.

[23] https://docs.hazelcast.com/imdg/4.2/cp-subsystem/cp-subsystem

Pod-2 participates in both groups. However, it retrieves and persists states only from *group-0*. Hazelcast provides several data structures for group members to use, including the *IAtomicReference*[24] class, which can be used to reference a *string* object. A request to persist a state is received by the group's leader. If the group member receiving the request is only a follower, it is forwarded to the leader. If most group members accepted and replicated the request, it is considered committed.

4.4.4 Failures of CP Members

Hazelcast removes crashed CP members from the cluster. However, it retains them in their CP Subsystem groups since it is unclear if a member encountered a failure or is not responding due to a network partition. Thus, they are still considered in the majority calculations of groups, which constitutes a threat to the availability of the cluster. Therefore, they need to be removed manually. In general, the Hazelcast cluster can tolerate failures of a minority of members, thus less than $n/2 + 1$ failures. In case a majority of cluster members crashes, the system needs to be restarted, losing all stored data in the process. Furthermore, a group cannot make any progress if a majority of its members is not responding, thus needs to be deleted. It can also be recreated; however, the progress is lost until this point.

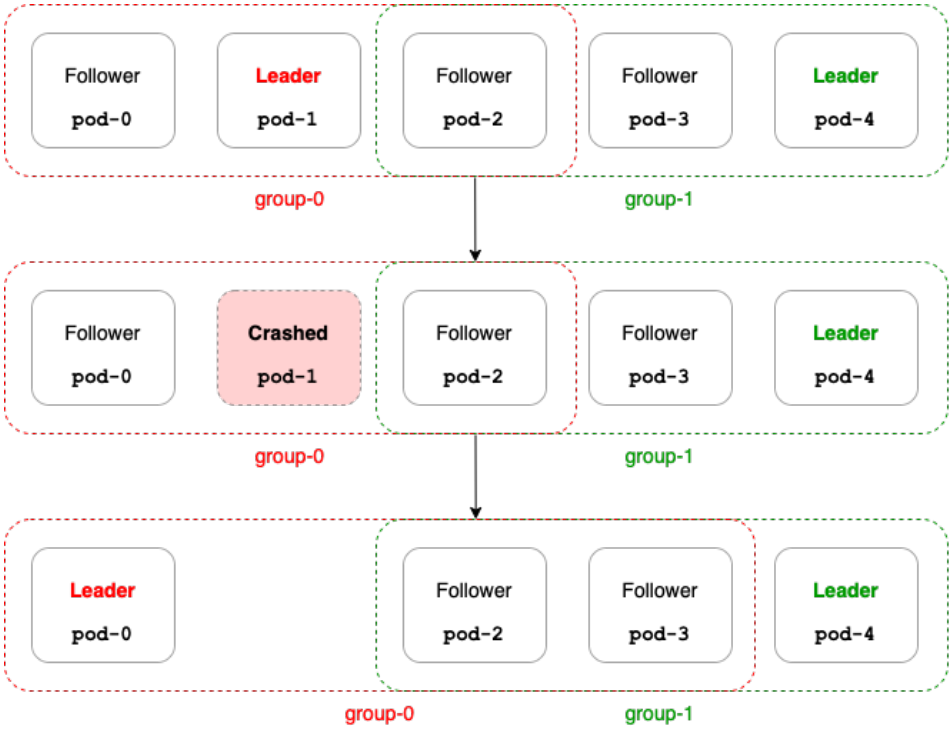


Figure 4.9: Group member is replaced after it crashed

[24] https://docs.hazelcast.org/docs/4.2/javadoc/com/hazelcast/cp/IAtomicReference.html

As depicted in Figure 4.9, *pod-1* crashed and is automatically removed from the cluster, and after the request is sent to remove it from the groups it participated in, it has completely vanished from the cluster. After Kubernetes redeploys the Pod, the respective Hazelcast instance reactivates the CP Subsystem and is added to the METADATA group. Meanwhile, *pod-3* takes its place and now participates in *group-0* and *group-1*. In case no other CP member is available, the majorities of each group are recalculated. Since *pod-1* was the leader of *group-0*, a new election round is started, which involves a small window of unavailability. A crashed member being the leader of multiple groups could constitute a bottleneck to the availability of the cluster. In case *pod-1* was a follower, the process of removing it from the group is not disruptive. Since *pod-1* is now part of the METADATA group, it can function as the replacement for future crashed members. A consensus replica is not required to be a member of a group to access the data structures within it. Thus, it is still possible for *pod-1* to process incoming requests and persist the state from its application replica.

4.4.5 Interaction with Stateful Application

At startup, the stateful replica *pod-0* attempts to retrieve the state from the consensus replica, as shown in Figure 4.10. Upon receipt, the latter determines the group based on the ordinal number of the identifier sent along. In case the group does not exist, it is created in the process, and *null* is returned. Otherwise, the state is retrieved from *group-0* and returned.

If the application replica *pod-0* receives *null*, i.e., no state was persisted with the fault-tolerant method, it assigns its identifier as the initial state. The state is subsequently updated by applying the hash function and sent back to the consensus replica. The latter determines the group based on the identifier of the application replica. Then, the state is persisted with the *IAtomicReference* object. If *consensus-0* is a follower in its group, the request is forwarded to the leader, which replicates the updated state to the other followers of the group once consensus is reached.

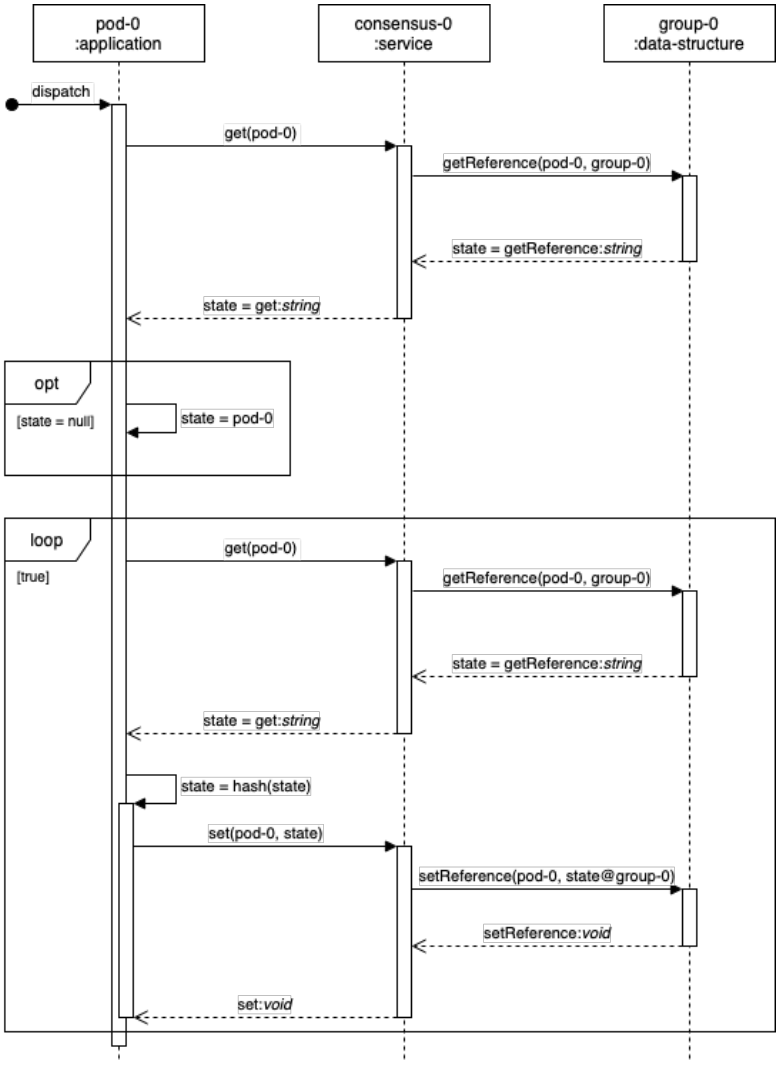


Figure 4.10: Interaction between application and decentralized fault-tolerant method

CHAPTER 5

Evaluation

The following chapter describes the evaluation process of the stateful application and the fault-tolerant methods. Section 5.1 outlines the relevant metrics concerning the proposed approaches' performance and scalability. Section 5.2 explains the process of evaluating both fault-tolerant methods. Finally, Section 5.3 discusses the results found during the evaluation.

5.1 Application Metrics

The stateful application collects various metrics during deployment, regardless of the integrated fault-tolerant method. The main goal of this evaluation process comprises two aspects: 1) a *comparison* of the *centralised* and *decentralised* approach in terms of their performance and scalability, and 2) examine the behaviour of the deployment in case a subset of Pods encounters a failure and crash. Therefore, the following parameters are collected and evaluated:

- **Persist Latency (ms)**: Period it takes to *persist* the state of an application replica with the integrated fault-tolerant method.
- **Retrieve Latency (ms)**: Period it takes to *retrieve* the state of an application replica from the fault-tolerant method.
- **Startup Delay (ms)**: Period until the fault-tolerant method is deployed, set up and ready to accept requests from the stateful application replicas.
- **Memory Usage (MB)** *Memory usage* of each Pod. In case of the centralized approach, it excludes the requirements of the CouchDB database.

The first two metrics are collected in a fixed time interval since each application replica periodically updates its state after retrieving it from the fault-tolerant method. *Memory usage* is collected throughout the deployment is running. In contrast, the *startup delay* is collected once a Pod is started, i.e., after it is initially deployed or restarted due to a failure. All the above-described parameters are provided to the *metrics* endpoint of the stateful application to be retrieved by the *Prometheus service* running on the Kubernetes cluster.

5.2 Evaluation Process

The evaluation process in this thesis comprises several different deployments to compare the performance of both fault-tolerant methods. Therefore, the stateful application and one of the two fault-tolerant methods are deployed with various replicas. In doing so, the *scalability* and *performance* of each method are being tested.

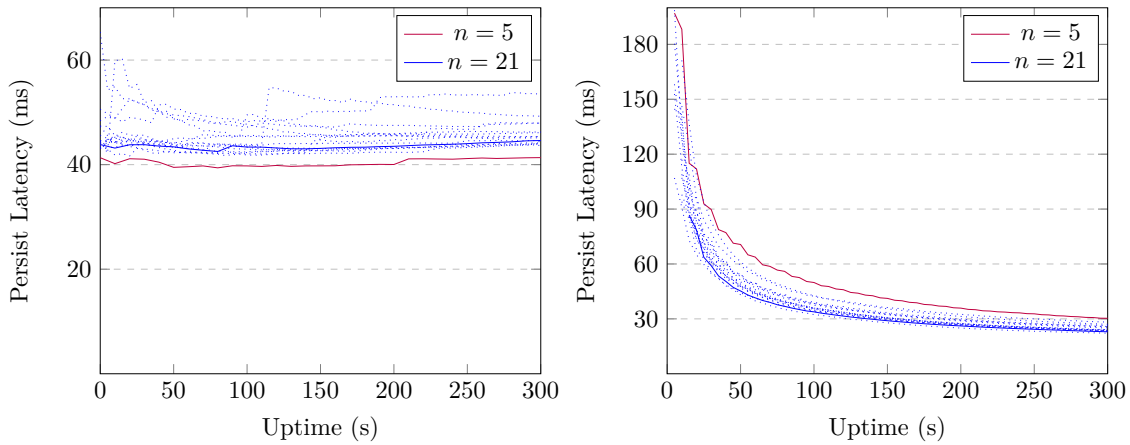
To streamline the deployment process, an additional *script* was implemented and made available on the public repository of this bachelor thesis. Kustomize[1] is used to generate deployment quickly manifests with different properties, e.g., the fault-tolerant method or the number of replicas, whereas the latter varies within the tests between 5 and 21. Furthermore, it allows us to define if Pods should crash during deployment to examine the fault-tolerant behaviour. This subset of Pods is selected randomly, whereas a maximum number of $f = \lfloor g/2 \rfloor$ Pods are allowed to crash - with g being the group size of the CP Subsystem. Suppose more than f Pods crash, the data consistency within a single group cannot be guaranteed. The centralised fault-tolerant method does not restrict the number of Pods that can crash; however, *two* Pods are selected to obtain comparable results. The failures are generated after one minute by deleting the respective Pods from Kubernetes. The evaluation period of all experiments amounts to *five minutes*. The script uses the Node.js client[2] to interact with the Prometheus service on the Kubernetes cluster, thus retrieving the aforementioned metrics once the evaluation period elapsed.

5.3 Results

The results concerning the deployments with 5 and 21 replicas are highlighted in the following plots. The latency observed while persisting application states varies hugely between both fault-tolerant methods, as depicted in Figure 5.1. The evaluation period follows the horizontal axis, whereas the *persist latency* is plotted along the y-axis. The centralised fault-tolerant method (5.1a) provides a consistent latency of 40 to 50 ms throughout the complete deployment, while single outliers can reach values up to 60 ms. This contrasts with the decentralised approach (5.1b), which shows an identical pattern for all observed values. In particular, they start at approximately 180 ms and eventually decrease over time to values below 30 ms. Figure 5.2 shows the results concerning

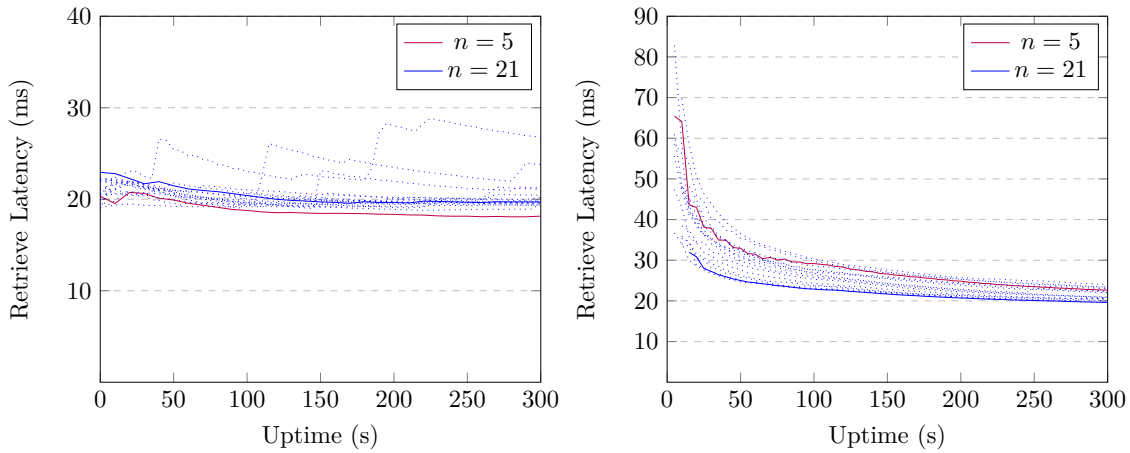
[1] https://kustomize.io
[2] https://github.com/siimon/prom-client

the *retrieve latency*, depicting a similar pattern as the aforementioned persist latency. Retrieving a state from the centralised approach (5.2a) generally takes between 18 to 23 ms; for some cases, it can take up to 28 ms. The maximum retrieve latency in case of the decentralized fault-tolerant method (5.2b) amounts to about 80 ms. However, it eventually decreases to values equivalent to those observed with the centralised fault-tolerant method over time.



(a) Centralized fault-tolerant method (b) Decentralized fault-tolerant method

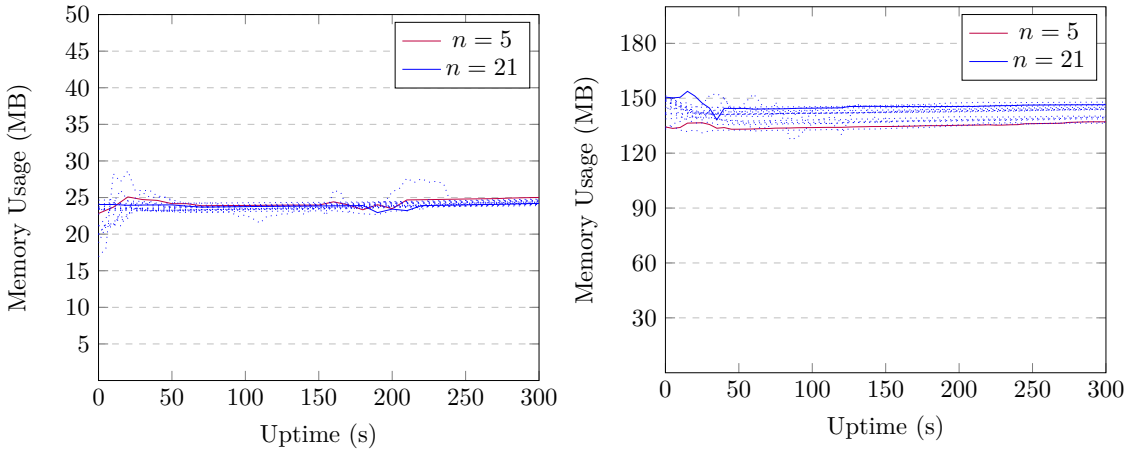
Figure 5.1: Comparison of average *persist latency* with scaling number of replicas



(a) Centralized fault-tolerant method (b) Decentralized fault-tolerant method

Figure 5.2: Comparison of average *retrieve latency* with scaling number of replicas

Figure 5.3 depicts the average memory usage of both fault-tolerant methods, whereas the evaluation period is plotted along the x-axis. The observed values marginally increase



(a) Centralized fault-tolerant method

(b) Decentralized fault-tolerant method

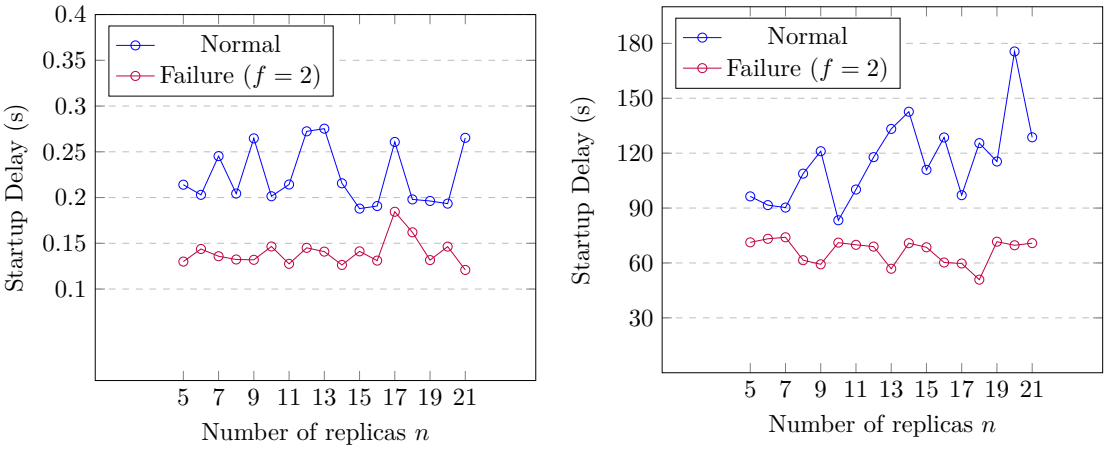
Figure 5.3: Comparison of average *memory usage* with scaling number of replicas

over time in both scenarios. However, they are fairly constant for this particular time frame. In general, the centralized method (5.3a) requires between 20 and 25 MB of memory, whereas the decentralized approach (5.3b) needs up to *six times* the memory requirements of the other method.

5.3.1 Observed Fault-Tolerance

As described above, a subset of Pods intentionally generates failures to examine the *fail-stop* behaviour in regards to performance and scalability of both fault-tolerant methods. Figure 5.4 shows the average *startup delay* for both methods, whereas one plot depicts the result of a regular operation, i.e., in which no failures occur, and another illustrates the observed values of deployments with the presence of Pod failures. The number of replicas n is plotted along the horizontal axis. The subset contains two Pods in both scenarios ($f = 2$), which are solely considered for the progressions marked in the colour purple. In doing so, the measurements are comparable with each other and show that both fault-tolerant methods remain operational even if two Pods crash simultaneously.

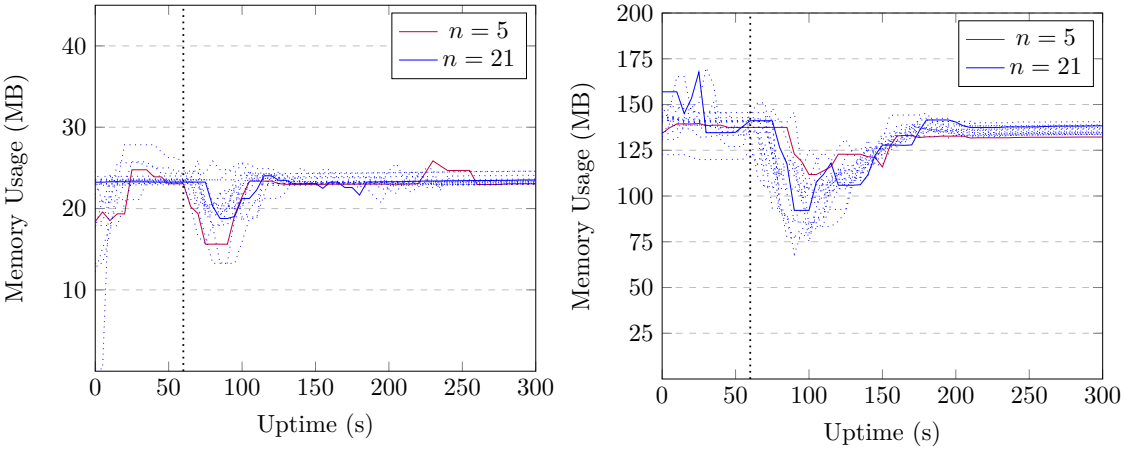
An application replica waits 0.22 seconds on average until the centralized fault-tolerant method (5.4a) is ready in normal operation. On average, the same instance must wait for 115.7 seconds until the decentralised method (5.4b) is ready to use, thus more than 700% longer. Both fault-tolerant methods can obtain improved results concerning the subset of Pods that intentionally crashed and are restarted. The restart period of the centralised method is about 37% shorter than the initial time it takes to deploy it. For the decentralised approach, this difference amounts to 42%. Those measurements do not include the time it takes for Kubernetes to deploy the respective Pods.



(a) Centralized fault-tolerant method

(b) Decentralized fault-tolerant method

Figure 5.4: Comparison of average startup delay with scaling number of replicas

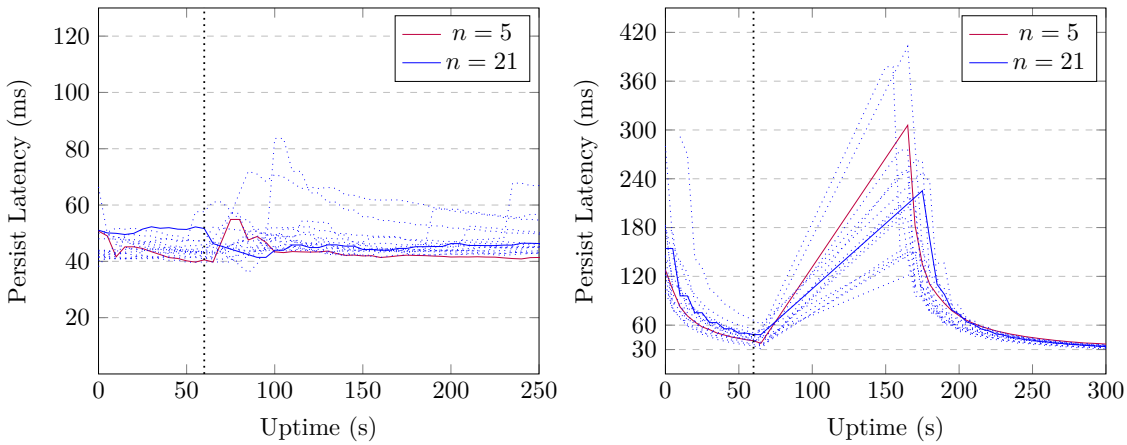


(a) Centralized fault-tolerant method

(b) Decentralized fault-tolerant method

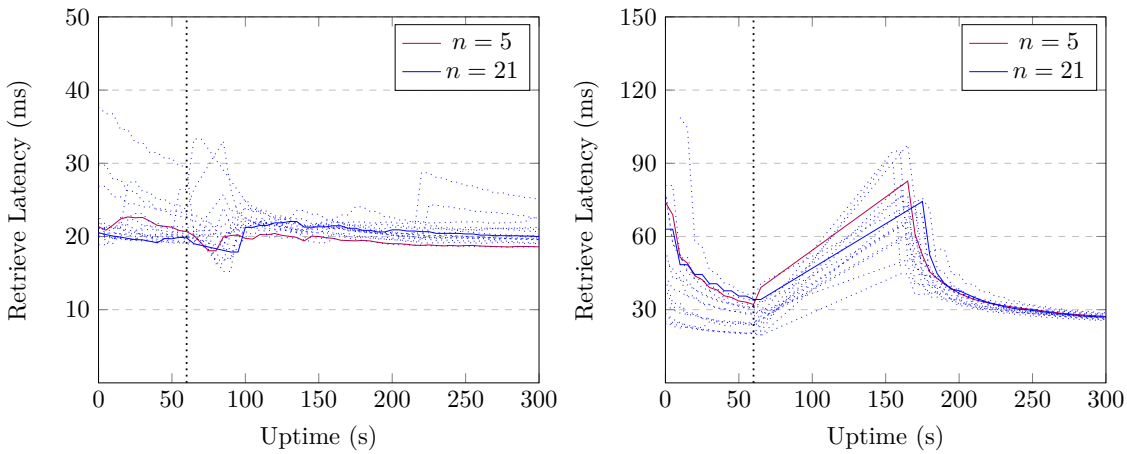
Figure 5.5: Comparison of average *memory usage* of crashed Pods

The average memory usage of the subset of crashed Pods is shown in Figure 5.5. The observed values decline right after the failures are generated, depicted by the horizontal line, the 60 seconds mark. In other words, the average memory usage of the crashed Pods decreases since their deployment is deleted. Upon resuming their normal operation, the requirements of both fault-tolerant methods even out at equivalent values to those observed before the Pod failures.



(a) Centralized fault-tolerant method (b) Decentralized fault-tolerant method

Figure 5.6: Comparison of average *persist latency* of crashed Pods



(a) Centralized fault-tolerant method (b) Decentralized fault-tolerant method

Figure 5.7: Comparison of average *retrieve latency* of crashed Pods

Figure 5.6 depicts the average *persist latency* of the crashed Pods. In the case of the centralised method (5.6a), the latency remains relatively the same for most of the deployments, regardless of the failures. However, values up to 80 ms can be observed for individual deployments. The results from the decentralised approach (5.6b) show a steep increase in the persist latency at the 60 seconds mark, reaching values up to 400 ms. The observations regarding the *retrieve latency* of the crashed Pods follow a similar pattern. Thus it does not strongly deviate from the expected behaviour in the case of the centralised method (5.7a). Furthermore, the decentralised approach results (5.7b) are observed to reach values up to approximately 100 ms after the failures are generated.

5.4 Discussion

Although the decentralised fault-tolerant method performs poorly at the beginning of the evaluation period, as apparent in the *persist* and *retrieve latency*, it eventually outperforms the centralised approach. The reason for this behaviour lies in the additional layers of the decentralised fault-tolerant method. The Spring Boot application and the Hazelcast library require more time to instantiate the underlying data structures than the CouchDB database. However, after the decentralised approach completed the initialisation of the needed resources, it performs better than the centralised method since it can rely on more optimised processes provided by the CP Subsystem. This component is specifically built for persisting and distributing stateful information - as opposed to the all-purpose CouchDB database.

Moreover, the decentralised method has higher memory requirements than the centralised approach. As previously mentioned, the demands of the CouchDB deployments are disregarded in the respective graphs. Further evaluation and exploration of this aspect can provide a more comprehensive picture of the memory requirements. The decentralised method also underperforms in terms of the observed startup delay. After a subset of Pods crash, it takes more time for the decentralised method to resume the operation of the crashed Pods. This observed latency hugely deviate from the values usually observed and is in line with measurements regarding the persist and retrieve latency, thus perform poorer due to the greater overhead of instantiating underlying data structures.

The number of replicas does not necessarily influence the performance of either of the two fault-tolerant methods. This behaviour is equivalent to the observations by *Netto et al.* [17]. Besides the varying number of replicas, their experiments also involve a scaling number of *clients*. Although they conclude that there is an increase in the latency observed by a client if additional replicas are added to the system, the case with only one client shows a reasonably constant progression. Thus, behaving similarly to the above described persist and retrieve latency.

CHAPTER 6

Conclusion and Future Work

The introduction of fog computing enables the deployment of virtualised services near end devices. The preferred method for bundling such applications is containerisation - a lightweight approach to isolating related services into a single package. Kubernetes allows the coordination of containerised applications' deployment and automates the maintenance of the applied components. However, external mechanisms are still required to restore application states if failures occur.

Therefore, this thesis compares the implementation and integration of two different fault-tolerant methods within a stateful application. A decentralised solution that involves SMR and the more traditional approach of incorporating centralised backups are discussed in the context of this work.

The literature review conducted in this bachelor thesis covers essential concepts regarding replicating stateful applications and summarises the findings of related works. Chapter 2 summarises important terminologies and provides important information about consensus algorithms, leader-based replication approaches and fault tolerance in general. In Chapter 3, relevant approaches of integrating SMR within Kubernetes are presented. The implementation of both fault-tolerant methods is provided in Chapter 4. The *decentralised* fault-tolerant method relies on SMR to achieve consensus and distribute stored data among healthy members within the network. This is opposed to the *centralised* method, which stores data in a single location and is accessed by all application instances.

Finally, the stateful application and integrated fault-tolerant methods were evaluated. This process involved the deployment of both methods with a varying number of replicas, whereas critical metrics were collected during this period. The first part of the evaluation compares the performance and scalability of the two approaches. The second part examines the behaviour of both approaches during the presence of intentionally generated failures concerning the metrics of the first part. The evaluation shows that the decentralised method involves more overhead which is apparent in more significant

memory usage and a higher startup delay. The latency observed during persisting and retrieving states eventually outperforms the centralised method as it can take advantage of the optimised functions provided by the Hazelcast library.

This thesis focuses on implementing and integrating two distinctly different fault-tolerant methods. When classified on a spectrum that reflects the degree of decentralisation, both approaches are to be found on either end. Thus, other solutions in-between the two proposed approaches can be implemented and integrated with the stateful application.

Although the number of replicas varies with the deployment of the decentralised fault-tolerant method, the number of related application instances remains unchanged. A *multi-leader* approach [10] would require further implementation and evaluation to allow multiple instances to modify the state data of an individual application replica.

The metrics collected for the evaluation part of this thesis provide details on the performance and scalability of the implemented fault-tolerant methods and stateful application. However, the influence of the allocation of other deployments on the cluster and the performance of Kubernetes are currently ignored. Further considerations can be made to provide more significant statements in this regard.

The Hazelcast library restricts the usage of *groups* of the CP Subsystem, as the current configuration dictates a single application replica to a specific group. Furthermore, it only allows $g/2$ group members to crash so that the underlying data structures remain intact (g being the group size). Any further considerations exceed the scope of this bachelor thesis; thus, optimising this aspect would enable the enhanced solution to be used for more use cases than the one described in this work.

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 28, 31

CAP consistency, availability, partition tolerance. 9, 10, 28, 32

HTTP Hypertext Transfer Protocol. 25, 28

JSON JavaScript Object Notation. 27

MVCC Multiversion Concurrency Control. 28

PBFT practical byzantine fault-tolerance. 5, 11, 12

PV Persistent Volume. 13–15

PVC Persistent Volume Claim. 13, 14

SC state controller. 15, 16

SMR state-machine replication. xi, xiii, 1–3, 5, 6, 10, 11, 13, 16, 45

TTL Time-to-live. 23, 27

URI Uniform Resource Identifier. 21

Bibliography

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, (New York, NY, USA), pp. 13–16, Association for Computing Machinery, 2012.

[2] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero, and M. Nemirovsky, "Key ingredients in an iot recipe: Fog computing, cloud computing, and more fog computing," in *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pp. 325–329, 2014.

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, pp. 299–319, Dec. 1990.

[4] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," *USENIX*, pp. 305–320, 01 2014.

[5] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[6] C. E. Bezerra, F. Pedone, and R. Van Renesse, "Scalable state-machine replication," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 331–342, 2014.

[7] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-paxos: Offloading the leader for high throughput state machine replication," in *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pp. 111–120, 2012.

[8] T. Ceolin, F. Dotti, and F. Pedone, "Parallel state machine replication from generalized consensus," in *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pp. 133–142, 2020.

[9] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998.

[10] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[11] A. Shahaab, B. Lidgey, C. Hewage, and I. Khan, "Applicability and appropriateness of distributed ledgers consensus protocols in public and private sectors: A systematic review," *IEEE Access*, vol. 7, pp. 43622–43636, 2019.

[12] A. Sousa, F. Pedone, R. Oliveira, and F. Moura, "Partial replication in the database state machine," in *Proceedings IEEE International Symposium on Network Computing and Applications. NCA 2001*, pp. 298–309, 2001.

[13] L. H. Le, C. E. Bezerra, and F. Pedone, "Dynamic scalable state machine replication," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 13–24, 2016.

[14] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, pp. 398–461, Nov. 2002.

[15] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, (New York, NY, USA), pp. 347–356, Association for Computing Machinery, 2019.

[16] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice based architecture: Towards high-availability for stateful applications with kubernetes," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pp. 176–185, 2019.

[17] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, and L. M. Sá de Souza, "State machine replication in containers managed by kubernetes," *Journal of Systems Architecture*, vol. 73, pp. 53–59, 2017. Special Issue on Reliable Software Technologies for Dependable Distributed Systems.

[18] Z. Bakhshi, G. Rodriguez-Navas, and H. Hansson, "Fault-tolerant permanent storage for container-based fog architectures," in *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*, vol. 1, pp. 722–729, 2021.