Kubernetes as an Availability Manager for Microservice Based Applications

Leila Abdollahi Vayghan

A Thesis

in the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

August 2019

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Leila Abdollahi Vayghan

Entitled: Kubernetes as an Availability Manager for Microservice Based Applications

and submitted in partial fulfillment of the requirements for the degree of

Master in Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. P. Rigby

_____ Internal Examiner
Dr. D. Goswami

_____ Internal Examiner
Dr. J. Rilling

_____ Co-Supervisor
Dr. F. Khendek

_____ Co-Supervisor
Dr. M. Toeroe

Approved by: _____
Dr. L. Narayanan, Chair
Department of Computer Science and Software Engineering

_____ 2019___

_____
Dr. Amir Asif, Dean,
Faculty of Engineering and Computer Science

ABSTRACT

Kubernetes as an Availability Manager for Microservice Based Applications

Leila Abdollahi Vayghan

The architectural style of microservices has been gaining popularity in recent years. In this architectural style, small and loosely coupled modules are deployed and scaled independently to compose cloud-native applications. Microservices are maintained and tested easily and are faster at startup time. However, to fully leverage from the benefits of the architectural style of microservices, it is necessary to use technologies such as containerization. Therefore, in practice, microservices are containerized in order to remain isolated and lightweight and are orchestrated by orchestration platforms such as Kubernetes. Kubernetes is an open-source platform that defines a set of building blocks which collectively provide mechanisms for orchestrating containerized microservices. The move towards the architectural style of microservices is well underway and carrier-grade service providers are migrating their legacy applications to a microservice based architecture running on Kubernetes. However, service availability remains a concern. Service availability is measured as the percentage of time the service is provisioned. High Availability (HA) is a non-functional requirement for service availability of at least 99.999%. Although the characteristics of microservice based architectures naturally contribute to improving the availability, Kubernetes as an orchestration platform for microservices needs to be evaluated in terms of availability. Therefore, in this thesis, we identify possible architectures for deploying stateless and stateful microservice based applications with Kubernetes and evaluate Kubernetes from the perspective of availability it provides for its managed

applications. Our experiment's results show that the healing capabilities of Kubernetes are not sufficient for providing high availability, especially for stateful applications. Therefore, we propose a State Controller which integrates with Kubernetes and allows for state replication and automatic service redirection to the healthy microservice instance. We conduct experiments to evaluate our solution and compare the different architectures from an availability perspective and scaling overhead. The results of our investigations show that our solution improves the recovery time of stateful microservice based applications by 55% and even up to 99% in certain cases.

Acknowledgments

First and foremost, I would like to thank God for giving me the strength, knowledge, and opportunity to undertake this research study. Without his blessings, this achievement would not have been possible.

I gratefully acknowledge my supervisors Dr. Ferhat Khendek and Dr. Maria Toeroe for their guidance, patience, and encouragements throughout my Master's study.

This work has been conducted within the NSERC/Ericsson Industrial Research Chair in Model-Based Software Management, which is supported by Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson and Concordia University.

I have great pleasure in acknowledging my gratitude to my colleagues at MAGIC, especially Dr. Mohammed Aymen Saied, for all their support and friendship.

I will forever be indebted for the love and support of my parents, my greatest role models. They have guided me towards the best possible life by teaching me to always love and always believe. And last, but not least, I am thankful to my loving husband. My dear Abbas, you stood beside me every step of the way. I will love and cherish you for the rest of my life.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the research domain and the motivations for this thesis followed by its contributions and organization.

1.1 Research Domain

With the advent of cloud computing [1], the microservices architectural style [2] has drawn a substantial amount of attention in the software engineering community. As opposed to monolithic architecture, the microservice based architecture tackles the challenges of building cloud-native applications that leverage the opportunities given by the cloud infrastructure [3].

Microservices [4] are a realization of the service-oriented architectural style for designing software composed of small services that can be deployed and scaled independently by fully automated deployment machinery and with minimum centralized management [5]. Each microservice has a separate business functionality, runs in its own process, and communicates through lightweight mechanisms often using APIs [2]. The fine granularity of this architectural style makes the scaling more flexible and efficient as each microservice can evolve at its own pace. Moreover, compared to monolithic applications, microservices are small and can restart faster at the time of upgrade or failure recovery. Microservices are loosely coupled and failure

of one microservice will not affect other microservices of the system. Because of these characteristics, adopting the architectural style of microservices can improve the service availability of applications [2].

Service availability is a non-functional requirement defined as the percentage of time a service is provisioned [6]. High availability is achieved when the system is available at least 99.999% of the time. Therefore, the total downtime allowed in one year for highly available systems is around 5 minutes [7].

To leverage the benefits of microservice based architectures, one needs to use technologies aligned with the characteristics of this architectural style. Containerization is the technology that enables virtualization at the operating system level [8]. Containers are lightweight and portable, therefore suitable for creating microservices. Docker [9] is the leading container platform that packages code and dependencies together and ships them as one container image. Since containers are isolated, they are not aware of each other. Thus, there is a need for an orchestration platform to manage the deployment of containers.

Kubernetes [10] is an open-source platform that enables the automated deployment, management, and scaling for containerized applications. Kubernetes alleviates the complexity of implementing applications' resiliency through its mechanisms for maintenance and healing. Therefore, it has become a popular platform for deploying containerized microservice based applications.

1.2 Thesis Motivation

The move towards the microservice based architectures is well underway. Organizations are migrating their legacy applications into cloud-native architectures by adopting the architectural style of microservices [11]. These microservice based applications are containerized and orchestrated by orchestration platforms such as Kubernetes. However, as an important quality attribute for carrier-grade service providers, service availability remains a concern. Some characteristics of microservices and containers such as being small and lightweight would naturally contribute to improving service availability [12]. Kubernetes provides healing for its managed microservice based applications [10]. The healing capability of Kubernetes consists of restarting the failed containers and replacing or rescheduling containers when their hosts fail. The healing capability also means not advertising unhealthy containers until they are ready again. These features would also improve the availability of the services provided by the applications deployed with Kubernetes. It is, therefore, important to evaluate the level of availability that Kubernetes can provide solely through its healing capabilities.

Replication is an important means of enabling availability. Stateless microservices are the most amenable to be replicated as they can be easily deployed as interchangeable instances. However, the same is not true for stateful microservices. Stateful microservices are not interchangeable and each one may have a unique state. This means one cannot bring stateful microservices down at a moment's notice and expect their services to be resumed by the other microservice instances. Deploying a replicated set of stateful microservices requires coordination of the different replicas to keep them synchronized and the "state" aspect makes orchestration even more complex.

The goals of this thesis are to evaluate Kubernetes as an availability manager for microservice based applications, identify weaknesses and propose solutions to improve the availability provided by Kubernetes.

1.3 Thesis Contributions

In this thesis, we identify the possible architectures for deploying microservice based applications with Kubernetes and qualitatively evaluate them from the perspective of service discovery and deployment. We also conduct availability experiments to measure the availability that Kubernetes can provide for its managed applications. Finally, we propose and evaluate a solution to improve the availability of stateful microservice based applications deployed with Kubernetes and we evaluate our solution from the perspective of availability and scaling overhead. The main contributions of this thesis are summarized as follows:

- Evaluation of the architectures for deploying microservice based applications with Kubernetes

  In this contribution we:

  - ✓ Identify the architectures for deploying stateless and stateful microservice based applications
  - ✓ Qualitatively evaluate the identified architectures
  - ✓ Conduct availability experiments under different failure scenarios and configurations to measure the achievable availability with Kubernetes
- A solution to improve the availability of stateful microservice based applications deployed with Kubernetes

  In this contribution we:

- ✓ Address the identified problems in managing the availability of stateful microservice based applications deployed with Kubernetes by proposing an availability architecture
- ✓ Introduce a "State Controller" component to integrate availability states with Kubernetes and assign active and standby roles to microservice instances
- ✓ Implement a prototype and conduct availability experiments under different failure scenarios and configurations to measure the achievable availability with our proposed solution
- ✓ Extend the State Controller prototype to assign availability states to multiple pods and conduct experiments to evaluate the achievable availability as well as the scaling overhead

1.4 Thesis Organization

This thesis is organized into five chapters. In Chapter 2, the background knowledge related to microservice based architectures, containerization, Kubernetes architectural components, and availability as well as related work are discussed. In Chapter 3, the possible architectures for deploying stateless and stateful microservice based applications with Kubernetes are identified and evaluated from the perspective of availability. In this chapter, the issues that Kubernetes faces in managing the availability of stateful applications are identified and discussed. In Chapter 4, we propose an architecture where we introduce a State Controller component which addresses the issues identified in Chapter 3. A prototype is implemented to measure, evaluate, and analyze the availability achievable with our solution. The proposed State Controller is modified in order to enable elasticity and evaluated in terms of availability and scaling overhead. Finally, in Chapter 5, we summarize our contributions and discuss potential future work.

Chapter 2

Background Information and Related Work

In this chapter, we present the related background information in Section 2.1 followed by the review of the literature for microservice based architectures and their availability in Section 2.2.

2.1 Background Information

In this section, we explain about the microservice based architectures [2] and their characteristics and discuss how containers can empower the usage of microservices [8]. We also introduce Kubernetes [10] and its objects as a platform for orchestrating containers. Lastly, we provide a general definition for service availability followed by an introduction to the Availability Management Framework (AMF) [13].

2.1.1 Microservices

The traditional way to create software is a monolithic approach. In this approach, the software is built as a large and single deployable unit to fulfill all business requirements. While the monolithic approach might be practical for small applications, a monolithic architecture for complex projects will create barriers for scalability and high availability [14]. Components of a monolithic application are tightly coupled which creates a “dependency hell” [15]. This leads to extended integration time and a lack of direct traceability to the source of errors during the

integration cycles. Moreover, because of the dependency, one cannot scale only a portion of a monolithic application as needed. Instead, the entire application needs to be scaled [2].

A microservice based architecture [16] is a realization of the service-oriented architectural style for developing software composed of small services (microservices) that can be deployed and scaled independently by fully automated deployment machinery, with minimum centralized management [16]. Microservices are built around separate business functionalities and a single microservice fulfills only one business requirement. Each microservice runs in its own process space and communicates with other microservices through lightweight mechanisms such as APIs [2]. Microservices of an application can be written in different programming languages and use different storage technologies and avoid "technology lock-in" [2].

Microservices' characteristics address the issues of monolithic architectures. For example, since each microservice implements a single business functionality, its code base will be small [2]. Therefore, maintaining and testing a microservice will require less effort. Also, microservices are loosely coupled and changing one microservice of the application does not require the whole system to be rebooted [2]. Moreover, because of the independency between microservices, it is possible to scale microservice based applications in a fine-grained manner. That is, increasing the number of one microservice instance while the number of other microservice instances of the application stays the same [2].

As we mentioned above, migrating towards a microservice based architecture has advantages. However, one needs to consider the disadvantages of this architectural style as well. For example, using microservices' architectural style means bringing the complexity of designing distributed systems into the design process [17]. Developers need to handle requests between microservices of the application and take the latency of remote calls into consideration.

Moreover, having multiple databases and managing transactions will require more effort [17]. Although testing a single microservice is less complex compared to a monolithic application, testing a microservice based application can be difficult. The reason is that all microservices of the application and their connectivity with the underlying infrastructure needs to be confirmed before testing the entire application [17].

2.1.2 Containers

In Sub-section 2.1.1, we discussed how the architectural style of microservices can address the issues of the monolithic approach. However, microservices architecture brings complexity for the developers that requires a certain level of automation and agility to help them adapt to this architectural style [14]. Today, containerization technologies are accelerating the use of microservice based architectures.

Containerization technology encapsulates the application's code and its dependencies and enables fine-grained resource control and isolation for them [18]. Containers implement virtualization at the operating system level. It is possible to run multiple containers on a single machine. These containers will share the OS kernel and run as isolated processes in user space [9]. On the other hand, Virtual Machines (VMs) implement virtualization at the physical hardware level which makes one single server to work as a number of servers [9]. Containers take less space (around tens of MBs) and are more lightweight compared to VMs. Because VMs include a full copy of an operating system and its binaries which can take tens of GBs.

Docker [9] is the leading container platform that encapsulates code and its dependencies together and ships them as a container image. Any machine that has the Docker container engine running can pull this image from the Docker Hub repository [19] and run containers based on this image. A Docker container image is a lightweight executable package of software that

includes the code, runtime, and system libraries to run an application. Container images become containers when they run on the Docker engine. They always run the same regardless of their infrastructure. Containers running on the same machine are isolated from each other and the application running inside a container will not impact other containers running on that machine.

The characteristics of Docker containers are aligned with the requirements of microservice based architectures [14]. For example, Docker containers are independently deployable units each of them providing a service. Moreover, they can be scripted to be created and launched and it is possible to automate their deployment and scaling. Each Docker container is an isolated environment that contains the required runtime for providing a particular service. Therefore, it is possible for each development team to use a different technology based on their needs and avoid the "technology lock-in" we mentioned before. Containers are lightweight and start up faster than virtual machines (VMs). Since one of the reasons microservices are designed small is fast restart in case of failure, a technology like containers should be used to avoid a bottleneck at start-up time. Moreover, containers running on a single machine run as isolated processes, therefore they will not affect each other or the underlying infrastructure. The isolation of containers helps to limit the failure impact of one microservice on other running microservices.

2.1.3 Kubernetes

In Sub-section 2.1.2, we explained how containers are a suitable solution for building microservices. We mentioned that containers isolate microservices from their environment. Because of this isolation, the containerized microservices of a microservice based application are

not aware of each other. Therefore, deployment of containers and their communication need to be orchestrated.

Kubernetes [10] is an orchestration platform that automates the deployment and management of containerized microservices. Kubernetes hides all this complexity behind its API. Therefore, Kubernetes' users do not need to implement the required mechanisms to manage their applications' resilience. Kubernetes' users only have to interact with the API to specify the desired deployment architecture and Kubernetes will be in charge of orchestration and availability management of the application. However, users with advanced requirements such as high availability may need to dive into Kubernetes details, since the Kubernetes architectural components can be used in different ways to deploy applications in a Kubernetes cluster.

The Kubernetes cluster has a master-slave architecture. The nodes in a Kubernetes cluster can be either virtual or physical machines. The master node hosts a collection of processes to maintain the desired state of the cluster. The slave nodes, which we will refer to simply as nodes, have the necessary processes to run the containers and also be managed by the master node.

An important process running on every node of a Kubernetes cluster is called the Kubelet. The Kubelet is a node agent that runs the containers assigned to its node via Docker and periodically performs health checks on them and reports to the master their status as well as the status of the node. Another node process is called the Kube-proxy that maintains network rules on the host and performs connection forwarding to redirect traffic to a specific container.

2.1.3.1 *Pods*

The smallest and simplest unit that Kubernetes deploys and manages is called a pod [10]. A pod is a collection of one or more containers. A pod is a process that provides an environment

to run containers by providing storage (called volumes) and network for them. A pod also has the specifications of how to run its containers. For example, it is possible to include in the pod's specification (called the pod template) to run a script inside the container once the pod is created. Customized labels can be assigned to pods to group and query them in the cluster. Containers in a pod share its IP address and port space. Once pods are created, Kubernetes assigns an IP address to them. In the pod template, it is specified which container is connected to which port of the pod. Therefore, any traffic received at the specified port of the pod will be redirected to its corresponding container. In practice, microservices are containerized and deployed on a Kubernetes cluster as pods. Pods can be created manually as well as by controllers. If a pod is created manually, Kubernetes will not monitor it for managing its lifecycle and will not recreate it if it fails. Therefore, it is recommended to deploy pods by using controllers. In the next subsection, we will introduce the controllers in Kubernetes' binary and how to deploy applications with them.

2.1.3.2 *Controllers*

Kubernetes' controllers deploy and maintain pods. A pod's template along with its desired number of replicas and other information such as upgrade strategy and pods' labels are included in a controller specification. Once the controller is deployed to the cluster, it creates the desired number of pods based on the provided template and continuously maintains their number equal to the desired number. For example, when a pod fails due to its container failure, the corresponding controller will automatically create a new one. In other words, controllers are watch loops that continuously work to bring the current state of the application to its desired state.

There are different types of controllers in Kubernetes and each of them is suitable for a specific purpose. For example, DaemonSet controllers run a copy of a pod on all nodes while

Job controllers create a number of pods and make sure they successfully terminate after they finish their tasks. Deployment controllers are mainly used for deploying stateless applications. On the other hand, StatefulSet controllers are used to manage stateful applications. A StatefulSet controller assigns a unique and persistent identity to each of its pods.

Stateful applications deployed as StatefulSet pods store their state in persistent storages. Kubernetes abstracts the details of storage solutions by providing two API resources: the Persistent Volumes (PV) and Persistent Volume Claims (PVC). A PV is a piece of storage in the cluster whose lifecycle is independent of those of the pods using it. PVs can be provisioned dynamically or statically. A PVC, on the other hand, is a request for storage made by a pod. A PVC binds the pod to a PV that matches the PVC's characteristics. A StatefulSet controller specification contains a PVC template which defines the characteristics of the PV (capacity and memory) that the pods need to be bound to once they are created. It is worth mentioning that a pod deployed by a Deployment controller can also store its data in a PV. However, that PV will be shared between all pods of the Deployment controller.

2.1.3.3 *Services and Ingress*

As the state of the cluster changes, a controller may delete a pod and move it in the cluster and cause the pod's IP address to change. Therefore, the pods' IP addresses are not reliable for communication. As mentioned before, Kubernetes allows to assign customizable labels to pods and select the pods based on these labels. Kubernetes also defines an abstraction called Service which selects pods as its endpoints list based on their labels. Services have static virtual IP addresses. The Kube-proxy watches the Kubernetes master and detects when a service or an endpoint is added or removed. For each service that is added, it installs iptable rules that redirect the traffic for the service's virtual IP and port to one of the service's endpoints. All requests

received at the IP address of the service are load balanced between the service endpoints in a random or round-robin manner. Although a pod gets a new IP address once it is deleted by a controller and created again, it will have the same labels as before. Because it is created based on the same pod template. Therefore, it will stay on the endpoints list of the service.

Kubernetes' services can be of different types. The default type is called "Cluster IP". Services of this type are accessible only from within the cluster. The "Node Port" type of service is built on top of a Cluster IP service and exposes the service on the same port of each node of the cluster. Lastly, a "Load Balancer" type of service is exposed externally only when the cluster is running in a public cloud.

Kubernetes provides another way, called ingress, to access services from outside of the cluster [10]. An ingress is a collection of rules for inbound connections to reach certain services in the cluster that are defined as backends for the ingress. For an ingress to work, an ingress controller needs to run in the cluster. Ingress controllers are not part of Kubernetes. To have an ingress controller, one should either implement it or use one that is available, e.g. Nginx [20] or HAProxy [21].

2.1.4 Service Availability

Service availability is an important non-functional requirement that defines the acceptable service outage in a period of time. Service availability is measured as the percentage of time a service is accessible in a given period [22]. The formula for measuring service availability is presented in Equation 2-1.

$$\text{Availability} = \frac{MTBF}{MTBF+MTTR} \quad (2\text{-}1)$$

In the Equation (2-1), MTBF is the Mean Time Between Failures which is the average mean time between two consecutive failures of a system, and MTTR is the Mean Time to Repair which is the average time to repair the system that has failed.

2.1.5 Availability Management Framework

The Service Availability Forum (SA Forum) [13] is a group of telecommunications and computing companies that cooperatively have standardized the high availability solutions. The SA Forum has defined several services and the Availability Management Framework (AMF) [13] is one of them. The AMF is a middleware service that provides availability for services provided by applications through coordinating redundant resources and performing recovery and repair actions. The AMF configuration is a specific organization of application resources that the AMF requires for managing the availability of services provided by the application.

Redundancy is an important mechanism for improving service availability. AMF defines five different redundancy models [13]. What follows is the explanation about these redundancy models defined by the AMF.

No-Redundancy Redundancy Model [13]**:** In this redundancy model, there are no standbys that protect the state of an active service provider instance. If an active instance fails,

service recovery will failover on any available spare service provider instance. However, the spare would not have the state of the failed service provider instance. If there are no spares available, service recovery will depend on the repair of the failed service provider instance.

N-Way Active Redundancy Model [13]**:** In this redundancy model, all service provider instances are considered active out of which N service provider instances can provide the same service. This model does not support standby assignments and is primarily used for stateless applications.

2N Redundancy Model [13]**:** In this redundancy model, one active service provider instance exists that provides a service and has one standby. Other service provider instances might be considered as spare (they do not have the state of the active). An active service provider instance serves the received requests while the standby service provider instance keeps the state of the service that the active service provider instance serves and is ready to take over if the active service provider instance fails. Note that in this model, a service provider instance cannot be active and standby at the same time.

N+M Redundancy Model [13]**:** This redundancy model is considered an extension to the 2N Redundancy Model. The N+M Redundancy model allows N service provider instances to have the active HA state and M service provider instances to have the standby HA state. Similar to 2N redundancy model, a service provider instance cannot be active and standby at the same time. Through using this model, it is possible to make better use of the resources. Because a standby service provider instance can be shared between multiple active ones.

N-Way Redundancy Model [13]**:** This redundancy model extends the N+M Redundancy model by allowing service provider instance to have active HA state for one service while having standby HA state for others.

2.2 Related Work

The architectural style of microservices has emerged primarily from the industry [5]. It is being adopted and investigated from different perspectives by practitioners and to a smaller extent by researchers in academia as well. In this section, we review the related work focusing on microservice based architectures and containers as their enabler. We also consider the related work for stateful microservices. In the end, we review other container orchestration platforms similar to Kubernetes.

Dragoni et al. in their work [2] propose the definition of a microservice as a small and independent process that interacts with other microservices by messaging. They define the microservice based architecture as a distributed application composed of microservices and discuss the impact of adopting the architectural style of microservices on the quality attributes of the application. Along with performance and maintainability, they specifically discuss availability as a quality attribute which is impacted by the microservice based architecture. Emam et al. in [23] found that as the size of a service increases, it becomes more fault-prone. Since microservices are small in size, in theory, they are less fault-prone. However, Dragoni et al. argue that at integration, the system will become more fault-prone on the integration level because of the complexity of launching an increasing number of microservices [2].

Other related works compare monolithic and microservice based approaches. For example, Villamizar et al. in [24] compare web applications deployed with a monolithic architecture and those deployed with a microservice based architecture. In their case study, they developed an application using the monolithic approach and one using the microservice based architecture and deployed both in a cloud infrastructure as a service. Their experiments show that the microservice based architecture can reduce the infrastructure costs by 17%. Although it is not

significant, the average response time of the microservice based solution was reported higher than that of the monolithic one. Another comparison of the monolithic approach with the microservice based architecture is done by Ueda et al. in [25]. They report 70% degradation of throughput with the microservice based architecture. In some cases, the Docker network configuration causes up to 33% degradation in performance [23].

Containers have been introduced as a technology that allows leveraging the benefits of microservices. Jaramillo et al. discuss in [14] how each Docker container is a deployable unit and an isolated box that contains the runtime environment and packages all dependencies within itself. For these reasons, Docker containers are suitable for microservices and bring automation, independency, and portability. Amaral et al. in [8] examine two different approaches where services can be developed as sets of containers: master-slave and nested containers. In the former, one container acts as parent and manages all other containers that work as peers. The nested containers model is inspired by Kubernetes' pod concept where the parent is a privileged container and the child runs in the parent's namespace. Their results show that the time to create a nested container is longer than that of a regular container and increases as the number of children grows.

Khazaei et al. in [26] propose a microservice platform for the cloud by using a Docker technology that provisions containers based on the requests of microservice users. One of the key differences between this platform and Kubernetes is that this platform has the ability to ask for more VMs from the infrastructure when needed while Kubernetes does not. Kang et al. in [27] propose a microservice based architecture and use containers to operate and manage the cloud infrastructure services. In their architecture, each container is monitored by a sidekick container and in case of failure, recovery actions are taken. They performed some experiments and concluded that recovering from container failure is faster than recovering from VM failure.

In their architecture, they have both stateless and stateful microservices. Their stateful microservice is a MySQL database with the active-active mode. For synchronizing data between microservices, they suggest shared storage and application level data replication. In the former, all MySQL microservices access the same data while in the latter the database process replicates the data across the cluster.

Netto et al. in [28] and [29] believe that Kubernetes is able to improve service availability of stateless applications. However, for stateful applications, Kubernetes faces some issues. In [28], to automate state replication between pods, all pod replicas execute the incoming requests. However, only the one that has received the request from the client will respond. In [29], they integrate a layer of containers between the client and the application (called Koordinator), which orders the requests received from the client and sends them to all application containers. They use a containerized firewall for redirecting the client requests to the Koordinator layer. This firewall is a single point of failure, especially in case of node failure. Moreover, although the availability is mentioned as one of the benefits in both works ([28] and [29]), it is not measured.

Soenen et al. in [30] aim to provide high availability for the management and orchestration (MANO) in the Network Function Virtualization (NFV) architecture by decomposing its functional blocks into microservices each performing a task in a workflow and interacting through remote calls over a network. To support availability, they deploy a redundant instance for each microservice type and both receive requests. Redundant microservices check each other's health through heartbeat. Meaning that the application needs to implement the availability logic. Moreover, each instance has the entire state and the task logs for both instances but only performs the tasks which was assigned to it. If a microservice instance does not receive a heartbeat, it also performs the tasks assigned to its peer. This can lead to a problem as the reason for

not receiving the heartbeat could be a network partition between two instances. In this case, there will be two instances performing the same task which can lead to data inconsistency.

[31] is an NFV specification with the goal of making virtualized network functions (VNFs) compatible with the cloud-native approach. It defines a set of requirements to make VNFs compatible with this approach. For example, VNF components should be containerized and also support high availability which is an important requirement of the cloud-native approach. Thus, redundancy at the VNF component (container) level must be supported as well. Moreover, since most telecom applications are stateful, the containers' state should be persistent and stored in external storages.

In Sub-section 2.1.5, we introduced the SA Forum [13] which defines a set of specifications to facilitate the development of carrier-grade applications. The Availability Management Framework (AMF) is one of the services defined by the SA Forum which is a middleware service that coordinates redundant resources and performs recovery and repair actions to provide availability for services provided by applications. The OpenSAF project [13] is an open-source middleware which implements SA Forum specifications including AMF and focuses on providing service availability for applications. The authors in [32] propose an elasticity engine that reacts to the fluctuations of the workload of applications managed by the AMF by modifying the AMF configuration resulting in a rearrangement of service provider resources. In their solution, they use the OpenSAF implementation of SA Forum specifications and demonstrate that elasticity can be managed at the application level within the AMF's framework that is for managing the availability of applications.

In Sub-section 2.1.3, we introduced Kubernetes as an open-source platform for orchestrating the lifecycle of containerized applications. However, there are other container orchestration

platforms available as well. For example, Docker Swarm [33] is a native clustering system for Docker that uses an API proxy system to turn a number of Docker hosts into a single virtual host. In this platform, a swarm is like a cluster in Kubernetes that is a group of virtual or physical machines with at least one master node. It is possible to use the Docker Engine command-line interface to create a swarm of Docker Engines. In Docker Swarm, the container images the swarm should use and the commands that need to be run in each container are defined in a service by a swarm administrator [34]. The Docker containers that execute the commands defined in the service are called tasks. When a master node assigns a task to a worker node, this task cannot be moved to another worker node. If a task fails, the master will assign a new version of that task to another worker node. Deploying applications with Docker Swarm is rather simple and Swarm mode is included in Docker Engine. However, one should note that it only supports Docker containers unlike Kubernetes that can use other container runtimes as well to run containers in pods (e.g., CRI-O [35], Containerd [36], and frakti [37]).

Another example of container orchestration platforms is Marathon [38] on Apache Mesos [39]. Apache Mesos is a cluster manager that simplifies resource allocation in public and private clouds by abstracting data center resources into one single pool of resources. It is able to scale both its underlying infrastructure and also the applications running on top of it. Apache Mesos can manage a diverse set of workloads including containerized applications. Marathon is the orchestration framework for managing containerized workloads that is built on top of Apache Mesos. Unlike Kubernetes that can run on any environment, Marathon can only run on Distributed Cloud Operating System (DC/OS) [40] and Apache Mesos. Apache Mesos has a master/slave architecture. The master node has information about the slave nodes' resources and sends this information to Marathon. A unit of work by Marathon is called a "task" that is scheduled on slave nodes based on the resource offers received from Mesos master.

Chapter 3

Microservice Based Architectures with Kubernetes and their Availability

In this chapter, architectures for deploying stateless and stateful microservice based applications with Kubernetes are identified and evaluated. In Section 3.1, we identify and evaluate these architectures qualitatively in terms of deployment and service discovery. In Section 3.2, we conduct experiments to quantitatively evaluate the stateless microservice based architectures from the perspective of availability and analyze the results in Section 3.3. We also address the availability challenges of Kubernetes in managing the availability of stateful applications in Section 3.3.

3.1 Architectures for Deploying Microservice Based Applications

There are different ways of using Kubernetes' architectural components to deploy applications. As an example, based on the application's characteristics, one can use a Deployment controller or a StatefulSet controller to deploy the application. Also, exposing the application to the world outside of the Kubernetes cluster or other services of the application inside the Kubernetes cluster can be done in different ways. For example, an application can be exposed either by using only services or an ingress. Moreover, the environment in which the Kubernetes cluster is running (e.g., public or private cloud) can affect the type of services used in deploying the application.

In this section, we present the possible architectures for deploying stateless and stateful microservice based applications with Kubernetes in different environments, i.e., public and private cloud. We also discuss service discovery and the related challenges for each case. These architectures are based on the Kubernetes architectural components described in [1].

3.1.1 Stateless Microservice Based Applications

The main controller for deploying stateless applications with Kubernetes is the Deployment controller. A Deployment controller specification is composed of a pod's template and a desired number of pods. After the Deployment controller is deployed, it creates the desired number of pod replicas and constantly works to bring the current state of the application to the desired state which means rescheduling pods when failures happen. In Kubernetes, a pod can store its data in a volume which is accessible by all of its containers. The volume is ephemeral and the data are lost when the pod is rescheduled or restarted. However, since stateless applications do not require the previously stored data for continuing their tasks, the loss of these data will not harm the application's functionality.

In Figure 3-1, a Deployment controller is used for deploying a stateless microservice based application. In this architecture, we consider a Kubernetes cluster composed of a number of VMs. Kubernetes runs on all VMs and creates a unified view of the cluster. One of the VMs is selected as the master and is in charge of managing the worker nodes. For simplicity, the application in this example is composed of only one microservice. The pod template for the containerized microservice as well as its desired number of replicas are included in the Deployment controller specification which is deployed to the cluster. It is possible to include customizable pod labels in a Deployment controller specification. The Deployment controller assigns these labels to pods when it is creating them. As seen in Figure 3-1, a

service is created which redirects the incoming requests to the application pods. The pods are selected as service endpoints based on the labels they have been assigned. There are

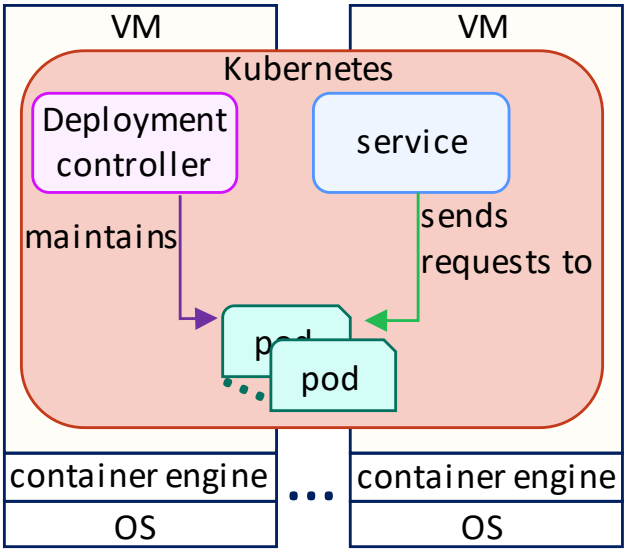

Figure 3-1. An architecture for deploying stateless microservice based applications with Kubernetes.

different types of services that can be used which we will discuss later in this sub-section.

3.1.2 Stateful Microservice Based Applications

In this sub-section, we bring the possible architectures for deploying stateful micro-service based applications with Kubernetes.

For stateful microservice based applications, Kubernetes provides different solutions. It is possible to deploy stateful microservice based applications with Deployment controllers as well as with StatefulSet controllers. In any case, the assumption is that the application's state data are stored in a persistent storage outside of Kubernetes called persistent volumes (PVs). Meaning that Kubernetes is not in charge of managing the PVs and it only consumes them.

3.1.2.1 *Deploying with StatefulSet Controllers*

The most common way of deploying stateful applications with Kubernetes is by a StatefulSet controller. In a StatefulSet controller specification, a PVC template is included along with the pod template. This PVC template describes the criteria (capacity, access mode, and etc.) for the PVs that the pods of the StatefulSet controller can be bound to. When the StatefulSet is deployed, a PVC is created for each pod binding it to a dedicated PV that meets these criteria. Since the data a pod stores in its PV are not shared with other pods of the StatefulSet, a mechanism such as sticky session is required to ensure that a client is always served by the same pod as its state is stored in the PV only accessible by that pod.

Figure 3-2 shows the architecture for deploying stateful applications using a StatefulSet controller. As shown in Figure 3-2 StatefulSet pods' names are a combination of their controller's name ("MS") and an ordinal index (MS-0, MS-1… MS-(n-1)). A difference that StatefulSet pods have compared to pods managed by other controllers is that they have persistent

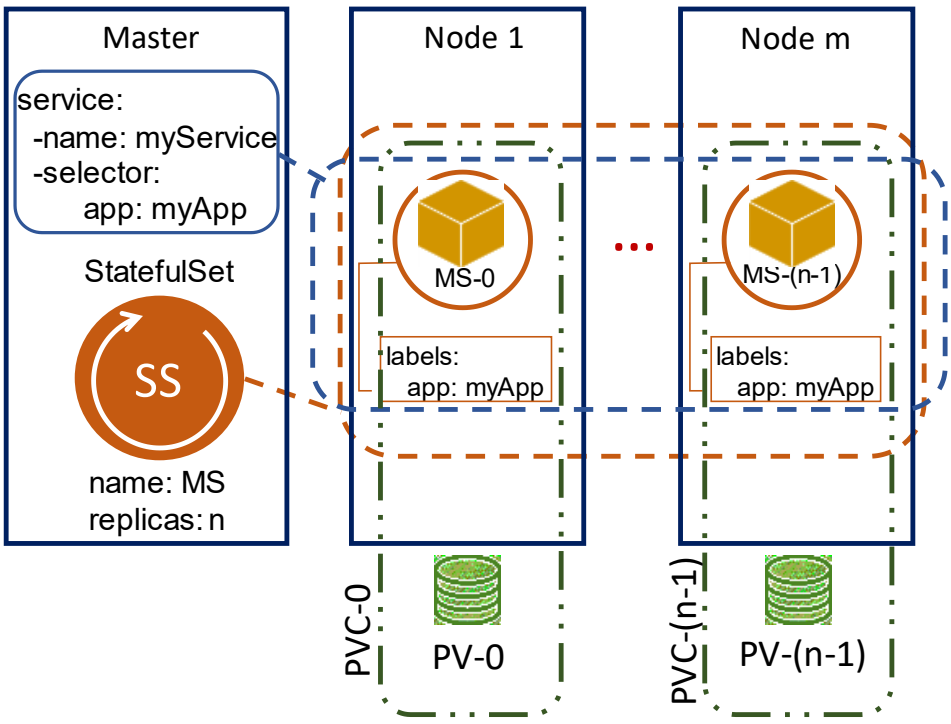

Figure 3-2. An architecture for deploying stateful microservice based applications using a StatefulSet controller.

identities. Meaning that if MS-0 which stores its state data in PV0 fails, the StatefulSet controller will restart the pod and will give the same identity to it. Therefore, MS-0 will be bound to PV0 again and it will have access to its state data stored prior to its restart.

3.1.2.2 *Deploying with Deployment Controllers*

Although StatefulSet controllers are the most commonly used controllers for deploying stateful applications, one can use Deployment controllers for this type of application as well. Similar to StatefulSets, the stateful Deployment pods can store their state data in a PV. However, with Deployment controllers, all pods have to share a PV. The reason is that it is not possible to include a PVC template in a Deployment controller specification. Therefore, one PVC should be created before deploying the application which will be used by all pods once they are deployed by the Deployment controller.

Figure 3-3 shows the architecture for deploying stateful applications using a Deployment controller. In this architecture, the state data for each client are shared between all pods. Therefore, all pods can serve a client as they have access to its data.

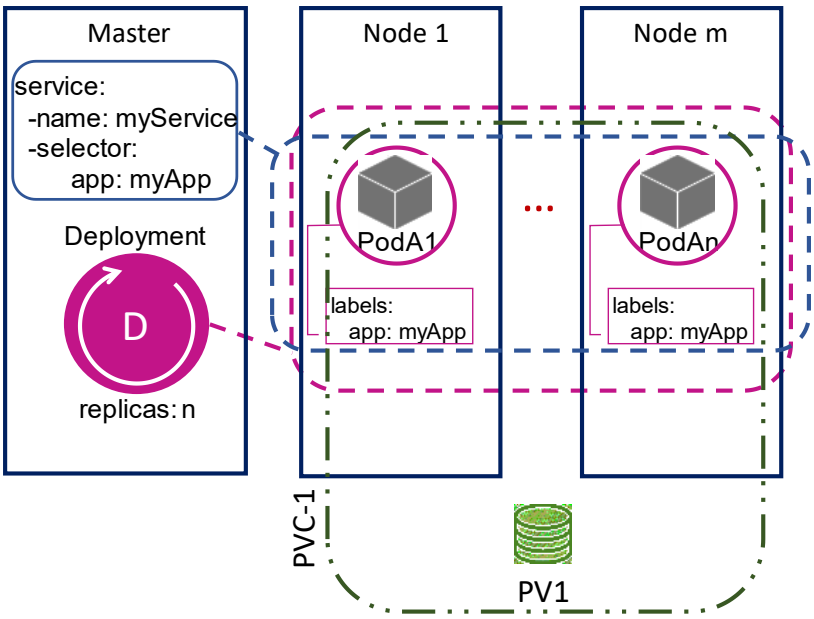

Figure 3-3. An architecture for deploying stateful microservice based applications using a Deployment controller.

3.1.3 Service Discovery

In the previous sub-section, we explained the possible architectures for deploying both stateless and stateful microservice based applications with Kubernetes. In all discussed architectures, we see that there is a service exposing the application. Kubernetes' services can have different types and depending on the application and the environment in which the Kubernetes cluster is running, a specific service type may be used. In this sub-section, we describe two ways of exposing applications deployed with Kubernetes: services and ingress. The method of service discovery does not depend on the type of controller used for maintaining the pods. Therefore in all examples, we only consider the architecture with a Deployment controller for stateless applications.

3.1.3.1 *Using Services to Expose Applications*

The most common way to expose applications deployed with Kubernetes to the world outside of the cluster or other services of the application is Kubernetes' services. Services can be of different types and are rules that are added to the IP tables of Kubernetes' cluster nodes.

The default service type is called the "cluster IP" which is only useful when we need to expose a set of pods to another service inside Kubernetes' cluster. However, exposing the application to the outside of Kubernetes cluster can be challenging.

A Kubernetes cluster can run in a private cloud as well as a public one. For each type of environment, a different service type should be used to expose the application to the outside world. For applications deployed with Kubernetes running in a public cloud, a service of type "Load Balancer" can be used as shown in Figure 3-4. In addition to a cluster IP, services of type "Load Balancer" have an external IP address that is automatically set to the cloud provider's load balancer IP address. Clients from outside of the cluster can access the application

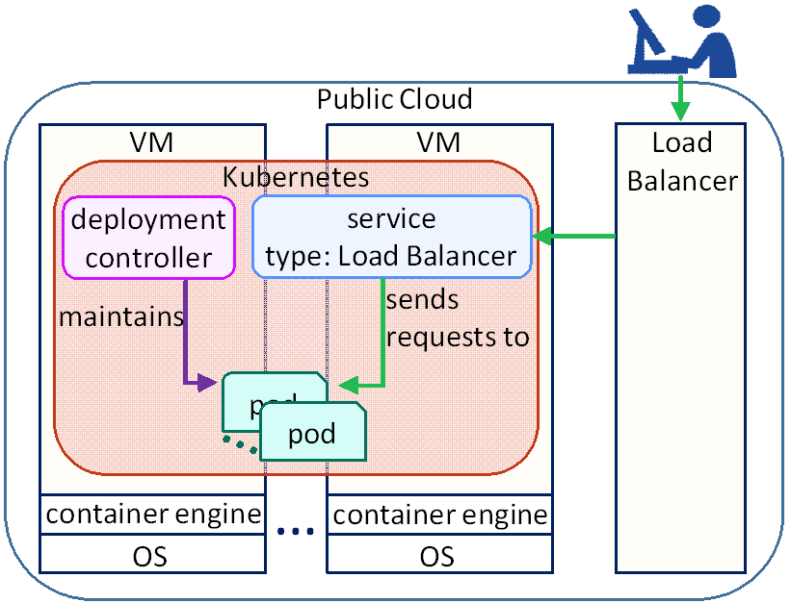

Figure 3-4. Public cloud - exposing services via services of type "Load Balancer".

through this public external IP address. The requests received at this address are redirected to the cluster IP of the service and later load balanced between all pods.

Services of type "Load Balancer" are a straight forward way to expose applications as they automatically obtain the cloud provider's load balancer IP. However, this feature does not exist in Kubernetes clusters running in a private cloud. Figure 3-5 depicts the architecture for exposing the applications deployed with Kubernetes running in a private cloud to the outside of the cluster. In this architecture, a service of type "Node Port" is used. This type of service exposes the application on the same port on every node in the cluster. The "Node Port" service is also built on top of a "Cluster IP" service. Meaning that all requests received at the specified port will be redirected to the cluster IP of the service and later load balanced between all pods. However, since it is not a good practice to expect the users to connect to the nodes directly, an external load balancer is used which distributes the requests between the nodes and delivers them to the port on which the "Node Port" service is exposed. The problem with this architecture is that for each service of the application in the cluster that needs to be exposed externally, we will need one external load balancer.

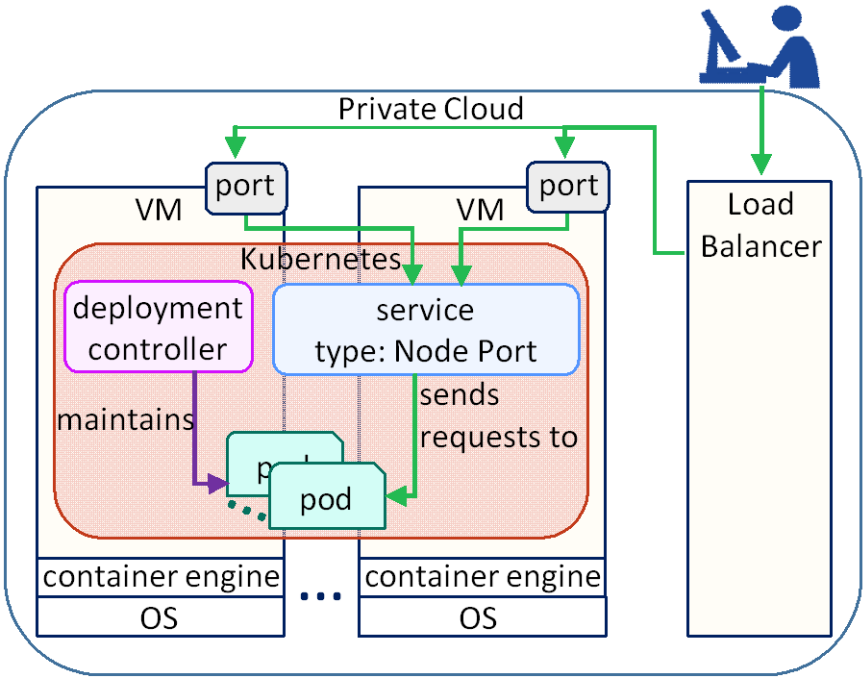

Figure 3-5. Private cloud - exposing services via services of type "Node Port".

3.1.3.2 *Using Ingress to Expose Applications*

Applications can have more than one service that need to be exposed externally and with the methods explained in the previous sub-section, we need one load balancer for each service. On the other hand, Kubernetes' ingress resource can have multiple services as backends and minimize the number of load balancers utilized. With an ingress resource, each service in the cluster can be given an externally reachable URL. An ingress controller should be deployed in the cluster in order to redirect the incoming requests to the backend services based on the rules defined in the ingress resource. The ingress controller is deployed as one pod using a Deployment controller.

In a Kubernetes cluster running in a public cloud, the ingress controller is deployed by a Deployment controller and exposed by a service of type "Load Balancer" (Figure 3-6). Therefore, requests for all services that are sent to the cloud provider's load balancer are received by the ingress controller and redirected to the appropriate service based on the rules defined in the

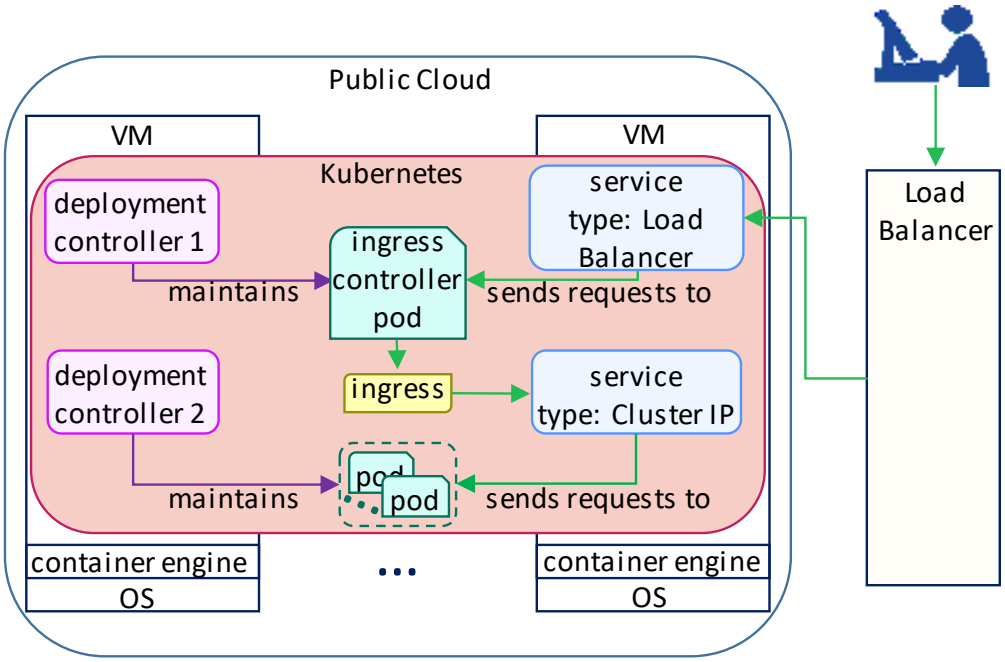

Figure 3-6. Public cloud - exposing services via ingress.

ingress resource. In a Kubernetes cluster running in a private cloud, the ingress controller is also deployed by a Deployment controller but exposed by a service of type "Node Port" (Figure 3-7).

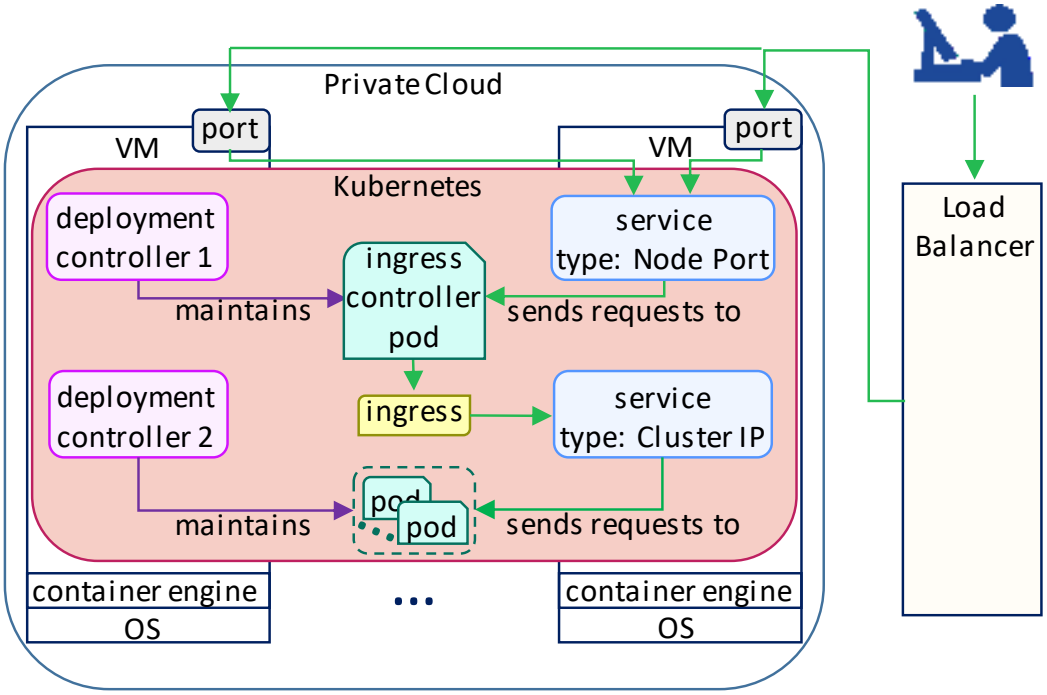

Figure 3-7. Private cloud - exposing services via ingress.

3.2 Availability of Stateless Architectures

For carrier-grade service providers, availability is an important non-functional requirement measured as the total outage time over a given period [6]. As these service providers are migrating towards the microservice based architecture, it is important to evaluate Kubernetes from the perspective of availability it can provide for its managed applications. In this section, we evaluate the architectures presented in Sub-section 3.1.1 for stateless microservice based applications from the perspective of availability by addressing the following research questions (RQ):

RQ1: What is the level of availability that Kubernetes can provide for its managed microservices solely through its repair actions?

RQ2: What is the impact of adding redundancy on the availability achievable with Kubernetes?

RQ3: What is the availability achievable with Kubernetes under its most responsive configuration?

RQ4: How does the availability achievable with Kubernetes compare to existing solutions?

To address these research questions, we conducted some availability experiments covering a number of failure scenarios and measured the defined availability metrics. These failure scenarios and the availability metrics are explained in the next sub-section.

3.2.1 Availability Metrics and Failure Scenarios

The availability metrics and failure scenarios for our experiments are as follows.

3.2.1.1 *Availability Metrics*

The metrics we use to evaluate Kubernetes from availability perspective are defined below. In Figure 3-8, we summarize the relations between these metrics.

Reaction Time: The time between the failure event we introduce and the first reaction of Kubernetes that reflects that the failure event was detected.

Repair Time: The time between the first reaction of Kubernetes to the failure event and when the pod failed due to the failure event is repaired.

Recovery Time: The time between the first reaction of Kubernetes to the failure event and when the service is available again.

Outage Time: The duration for which the service was not available. It represents the sum of the reaction time and the recovery time as shown in Figure 3-8.

3.2.1.2 *Failure Scenarios*

Kubernetes offers three levels of health check and repair action for managing the availability of the deployed applications. First, at the application level, Kubernetes ensures that the

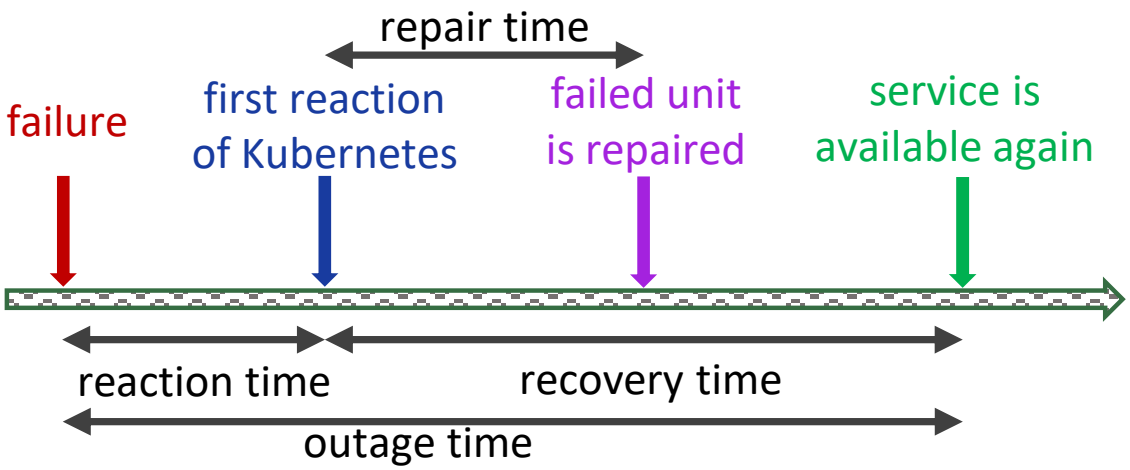

Figure 3-8. Availability metrics.

software components executing inside a container are healthy either through process health check or predefined probes. In both cases, if the Kubelet discovers a failure, it will react according to the defined restart policy. Second, at the pod level, Kubernetes monitors the pod process failures. That is, monitoring the pod process that is the environment provided for running application containers by providing shared storage and network for them. Finally, at the node level, Kubernetes monitors the status of the cluster nodes through its node controller component. If the node hosting a pod fails, the pod is rescheduled on another healthy node. With respect to these levels of health check, we defined three failure scenarios which are explained as follows.

Service Outage Due to Application Container Failure: In this scenario, the failure is simulated by killing the application container process from the OS.

Service Outage Due to Pod Process Failure: When a pod is deployed, along with the application containers specified in its template, one extra container is created which is the pod process. Since the pod itself is a process in the OS, it is possible that it crashes. In this scenario, the failure is simulated by killing the pod process from the OS.

Service Outage Due to Node Failure: In this scenario, a node that is hosting a pod fails. For some experiments, this scenario is simulated by Linux's reboot command and for others is simulated by shutting down the node.

In the following Sub-sections, we address the previously posed research questions for stateless microservice based applications deployed with Kubernetes as well as stateful ones.

In this Sub-section, we present the concrete architectures, the experiments, the results and the analysis for answering the research questions we brought before for stateless micro-service based applications deployed with Kubernetes.

For these experiments, we set up a Kubernetes cluster in a private cloud (Figure 3-7). This cluster is composed of three VMs running on an OpenStack cloud. Ubuntu 16.04 is the OS running on all VMs. Kubernetes 1.8.2 runs on all VMs and the container engine is Docker 17.09. The Network Time Protocol (NTP) [41] is used for time synchronization between the nodes. The deployed application is VideoLan Client (VLC) [42]. There is one container image in the pod template, in which VLC is installed. Once a pod is deployed, an application container will be created based on this image and will start streaming from a file. The application is stateless and if the container is restarted, it will start streaming from the beginning of the file.

3.2.2 Evaluating the Availability of Stateless Applications Deployed with Kubernetes

In this sub-section, we evaluate the availability that Kubernetes can provide for the stateless applications under different scenarios and address the research questions posed earlier in this section.

3.2.2.1 *Evaluating the Repair Actions with the Default Configuration of Kubernetes (RQ1)*

The common practice to evaluate Kubernetes' repair actions is to simulate failures through administrative operations (e.g. delete the pod or the node) using the Kubernetes command-line interface (CLI) and then observe how fast a new pod replaces the failed one [43]. Due to the use of Kubernetes' administrative operations, such a failure is not a spontaneous

event that Kubernetes needs to detect and react to. Instead, the operation is executed by Kubernetes in due order often in a graceful manner. Therefore, these operations cannot reflect common execution failure scenarios, which are anything but graceful and happen spontaneously maybe as a result of external failure events (e.g., process or physical node crash). Drawing conclusions based on such administrative operations would not be accurate. Hence, it is important to identify and simulate execution failure scenarios due to external events properly and measure the availability in these cases before making conclusions.

In this sub-section, we first bring the experiments where service outage is due to external events causing execution failures. Then, we repeat the failure scenarios where the failure is injected by administrative commands. We analyze the results and compare the measured availability of experiments with failure events due to administrative Kubernetes operations to those with external execution failure events.

Figure 3-9 shows the architecture for these experiments. For this research question, we are interested in measuring the availability that Kubernetes can provide only through its repair actions. Therefore, the redundancy model in this case is No-Redundancy without spare [13] which means the number of pods in the Deployment controller specification is only one. The measurements of the experiments with external execution failure events are shown in Table 3-1 and those with administrative failure events are shown in Table 3-2. What follows is the detailed explanation for each failure scenario in these experiments.

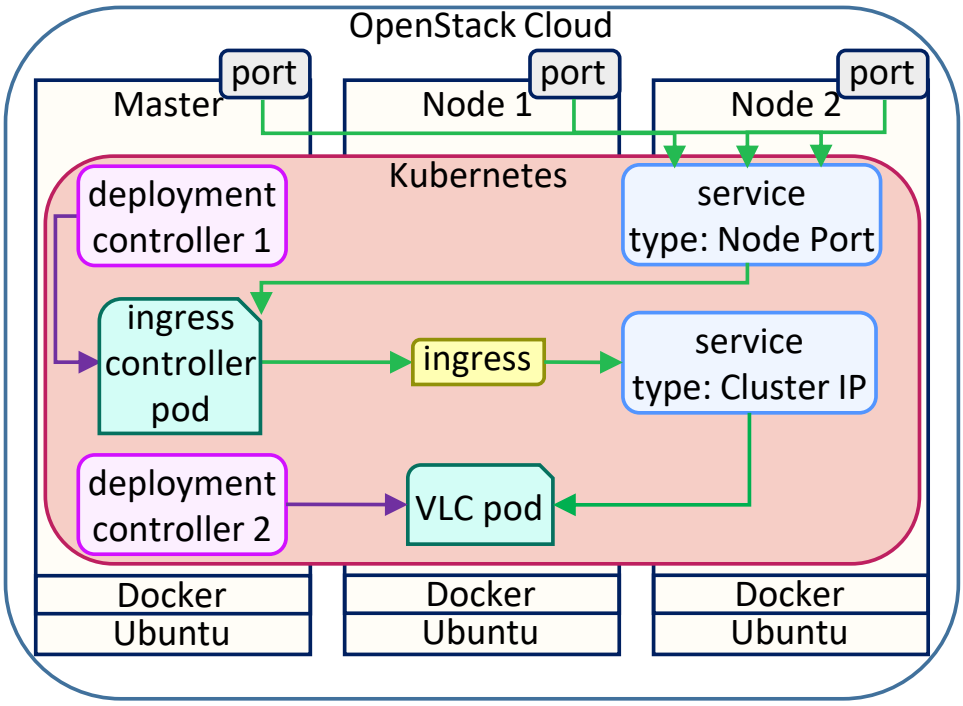

Figure 3-9. Concrete architecture for experimenting with Kubernetes - Stateless microservice based application with No-Redundancy redundancy model.

Service Outage Due to External Execution Failure Events

We conducted availability experiments where the failures are external execution failures resulting in service outage which can reflect the real-life failure events.

Service Outage due to VLC Container Process Failure: In this scenario, the failure is simulated by killing the VLC container process from the OS. When the VLC container crashes, the Kubelet detects the crash and brings the pod to a state where it will not receive new requests. At this time, that is the reaction time, the pod is removed from the endpoints list. Later, the Kubelet restarts the VLC container and the video will start from the beginning of the file. This time marks the repair time. Recovery time is when the pod is in the endpoints list again and is ready to receive requests.

Service Outage due to Pod Process Failure: In this scenario, the failure is simulated by killing the pod process from the OS. When the pod process is killed, the Kubelet detects that the pod process is no longer present and this marks the reaction time. When the new pod is created and its VLC container is started, the video will start streaming from the beginning of the file and we consider the pod as repaired. After, the Kubelet will add the new pod to the endpoints list and it will be ready to receive new requests. This marks the recovery time.

Service Outage due to Node Failure: In this scenario, node failure is simulated by the Linux's reboot command on a VM hosting the pod. As mentioned before, the Kubelet is responsible to report the status of the node to the master, and it is the node controller of the master who detects the failure of the node. With the default configuration of Kubernetes, when a node hosting a pod fails, it stops sending status updates to the master and the master will mark the node as not ready after the fourth missed status update. This time is the reaction time. When the node is marked as not ready, the VLC pod on the node is scheduled for termination and after it is completely terminated a new one will be created. The repair time is when the new VLC pod is started and streaming the video. Recovery time is when the pod is added to the endpoints list of the service.

Service Outage Due to Administrative Failure Events

As explained before, the common practice to evaluate Kubernetes' repair actions is by injecting "failure" events from Kubernetes' CLI. In order to compare with external execution failures, we conducted experiments that cover two failure scenarios. In the first scenario, the failure is simulated by deleting the pod administratively. In the second scenario, the failure is simulated by administrative deletion of the node. Note that since there are no commands in Kubernetes that can terminate the application container, we do not have the scenario where service outage is due to administrative application container failure.

Service Outage due to Administrative Pod Termination: In this scenario, the pod is ordered to terminate by a command and will consequently be removed from the endpoints list of the service and this is when we consider that the pod has failed. By default, pods have 30 seconds of graceful termination period. During this time, the pod will not receive new requests but will keep serving the requests previously assigned to it. This gives ample time to Kubernetes to schedule a new pod and deal with incoming requests. Since it is the responsibility of the deployment controller to always maintain one replica of this pod, it will bring up a new one and this event marks the reaction time. The repair time is when the new pod is started. Although at this time, the pod has started and is streaming the video, it will not be available to users unless it is added to the endpoints list of the service. Therefore, we consider the streaming service as recovered when the new pod is added to the endpoints list of the service and starts streaming.

Service Outage due to an Administrative Deletion of the Node: In this scenario, a node hosting a pod is deleted using a Kubernetes' CLI command. As a result, the cleanup of all containers and processes related to Kubernetes on this node is initiated. Any pod running on the node that is to be deleted enters a state that it does not receive new requests. Hence, this is what we consider the moment of failure. However, the behavior in this scenario is different from that of the administrative pod failure scenario. Here, the pod will serve the previously assigned requests for only around one second (not the default 30 seconds of graceful termination period). Shortly after, the pod is completely deleted and this time marks the reaction time. When the pod is completely deleted, the deployment controller will attempt to add a new pod

on another node. Repair time is when the new pod is started. Recovery time is marked later when the new pod is added to the endpoints list of the service and starts streaming.

Table 3-1. Experiments with Kubernetes under Default Configuration – External Execution Failures with No-Redundancy Redundancy Model.

failure trigger (unit: seconds)	reaction time	repair time	recovery time	outage time
VLC Container Failure	0.716	0.472	1.050	1.766
Pod Process Failure	0.496	32.570	31.523	32.019
Node Failure	38.187	262.542	262.665	300.852

Table 3-2. Experiments with Kubernetes under Default Configuration – Administrative Failures with No-Redundancy Redundancy Model

failure trigger (unit: seconds)	reaction time	repair time	recovery time	outage time
Administrative Pod Termination	0.041	0.982	1.547	1.588
Adiministrative Node Deletion	0.031	1.009	1.500	1.531

3.2.2.2 *Evaluating the Impact of Redundancy on the Availability (RQ2)*

In the previous sub-section, we evaluated the repair actions of Kubernetes in providing availability for its applications. However, an important mechanism for improving availability is adding redundant instances. In this sub-section, we investigate the impact of adding redundancy on the availability provided by Kubernetes. We consider the architecture of Figure 3-10 where the number of pod replicas that the Deployment controller maintains is increased to two. In this architecture, we have an N-Way Active redundancy model [13]. In this redundancy model, a number of microservice instances are deployed and since they are stateless, all of them are capable of providing the same service. We evaluate the availability metrics for each of the failure scenarios under the default configuration of Kubernetes with an N-Way Active redundancy model. We compare the results to the previous experiments (Sub-section 3.2.2.1).

The measurements for this set of experiments are shown in Table 3-3. What follows is the detailed explanation for each failure scenario in these experiments.

Service Outage due to the VLC Container Process Failure: In this scenario, similar to the No-Redundancy redundancy model, the reaction time is when the Kubelet detects the VLC container has crashed and removes the pod from the endpoints list. By just removing the unhealthy Pod1 from endpoints list, the service is recovered. This is because another healthy pod is still on the endpoints list and ready to serve the requests. Therefore, the reaction time for this scenario is the same as the recovery time. The repair time is when the Kubelet has restarted the crashed VLC container and the video has started streaming again.

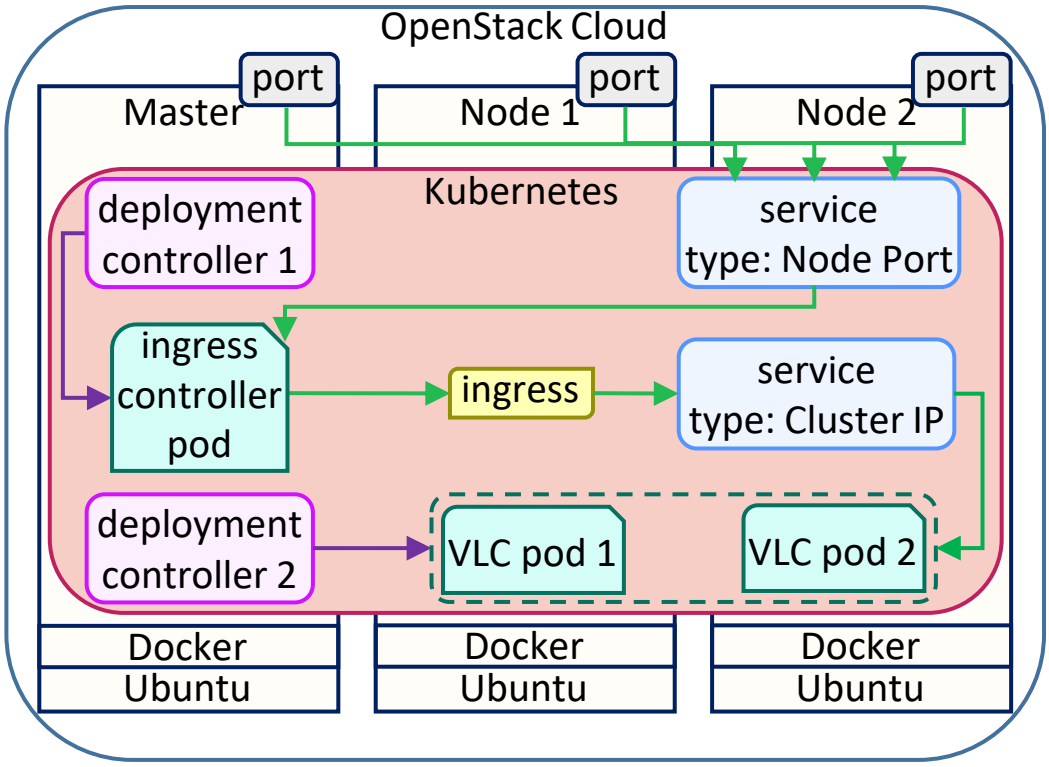

Figure 3-10. Concrete architecture for experimenting with Kubernetes - Stateless microservice based application with N-Way Active redundancy model.

Service Outage due to Pod Process Failure: In this scenario, the same as for the No-Redundancy redundancy model architecture, the reaction time is when the Kubelet detects that the pod is no longer there. Similarly to the previous scenarios, the recovery time is when the unhealthy pod is removed from the endpoints list making the healthy pod the only endpoint of the service. The repair time is when a new pod is created and its VLC container is started and streaming the video.

Service Outage due to Node Failure: In this scenario, node failure is simulated by the Linux's reboot command on a VM hosting the pod. The reaction time in this scenario is the same as for the No-Redundancy redundancy model architecture, i.e. the time the master marks the node as not ready and schedules the pod for termination. The recovery time is when the IP

of the failed pod is removed from the endpoints list. The repair time is when the failed pod is terminated and another one is created.

Table 3-3. Experiments with Kubernetes under the Default Configuration – N-Way Active Redundancy Model.

failure trigger (unit: seconds)	reaction time	repair time	recovery time	outage time
VLC Container Failure	0.579	0.499	0	0.579
Pod Process Failure	0.696	30.986	0.034	0.730
Node Failure	38.554	262.178	0.028	38.582

3.2.2.3 *Evaluating the Repair Actions with the Most Responsive Configuration of Kubernetes for Supporting Availability (RQ3)*

As observed in Sub-sections 3.2.2.1 and 3.2.2.2, the default configuration of Kubernetes has a significant impact on the service outage. Our analysis for the different failure scenarios has led to the identification of the aspects that need to be modified to reduce the observed outage. One aspect affecting the service outage is the graceful termination signal sent to the application container in the scenario of pod process failure which takes at least 30 seconds. For the node failure scenario, the frequency of node status posting by the Kubelet to the master is 10 seconds and the number of allowed missed status updates before marking a node as unhealthy is four which makes the reaction time between 30 to 40 seconds. Moreover, Kubernetes waits around 260 seconds to delete the failed pod and recreate a new one. All these factors hinder the availability of the application and therefore, we change them to measure the highest achievable availability with Kubernetes.

To answer this research question, we perform two sets of experiments where Kubernetes has the most responsive configuration. In the first set, for the pod process failure, the configuration parameter for the graceful termination of pods is set to zero seconds. In the second set, for the node failure, the configuration parameters related to handling node failure are set to the lowest value possible (one second). We are aware of the network overhead and potential false-positive node failure reports for the most responsive configuration. However, our goal in this experiment is to measure the best achievable availability when deploying applications with Kubernetes. These experiments were conducted with both No-Redundancy and N-Way Active redundancy model architectures (Figure 3-9 and Figure 3-10). The results of these experiments are presented in Table 3-4 and Table 3-5. What follows is the detailed explanation for each reconfiguration.

Reconfiguring the Graceful Termination Period of Pods: As it was mentioned, when a pod process fails, a graceful termination signal is sent to Docker to terminate the application container which delays the repair of the pod for 30 seconds. In the No-Redundancy redundancy model, this grace period affects the recovery time, because a new pod will not be created unless the failed one completely terminates. To reduce this grace period, we updated the pod template and set the grace period to zero. We repeated the experiments for the pod process failure scenario and evaluated the impact of this change on service outage.

Reconfiguring Node Failure Handling Parameters: To have the most responsive Kubernetes configuration, we reconfigured the Kubelet of each node to post the node's status every second to the master. The node controller of the master was also reconfigured to read the updated statuses every second and allow no missed status updates for each node. We repeated the experiments for the node failure scenario in order to evaluate the impact of this reconfiguration on service outage.

Table 3-4. Experiments with Kubernetes with changed configuration - service outage due to pod container failure.

redundancy model (unit: seconds)	reaction time	repair time	recovery time	outage time
No-Redundancy	0.708	3.039	3.337	4.045
N-Way Active	0.521	3.008	0.032	0.554

Table 3-5. Experiments with Kubernetes with changed configuration - service outage due to node failure.

redundancy model (unit: seconds)	reaction time	repair time	recovery time	outage time
No-Redundancy	0.976	2.791	2.998	3.974
N-Way Active	0.849	2.173	0.022	0.872

3.2.2.4 *Comparing Kubernetes with Existing Solutions for Availability Management (RQ4)*

To better position the availability results obtained with Kubernetes, we address RQ4. AMF [13] is a standard middleware service for managing the availability of components based applications. It has been implemented, with other middleware services, in the OpenSAF middleware [13], a proven solution for availability management. In a previous work [44], a set of experiments for different failure scenarios with the same application (VLC) was conducted with OpenSAF. The architecture for the experiments with OpenSAF is shown in Figure 3-11. In AMF, a component is the smallest service provider entity and the resources represented by the component encapsulate specific application functionality. In the architecture of Figure 3-11, a VLC component and an IP component are used. Components are either SA-Aware or Non-SA-Aware. SA-Aware components implement AMF API and register with AMF to manage service availability [13]. SA-Aware components are primarily used for stateful applications. The Non-SA-Aware components do not interact with AMF directly and AMF is only in charge of managing their lifecycles. In these experiments, the VLC component that is used is not modified and it is a Non-SA-Aware component.

Moreover, in this architecture, the SU stands for service unit which is a cooperation of components combining their individual functionalities to provide a higher level service. The SU in AMF is the unit of redundancy. Also, CSI stands for component service instance and is an abstraction of a service provided by a component that is assigned to the component by AMF at runtime. The SI (service instance) is an aggregation of CSIs. The SI is a single workload assigned to a SU. It is possible to assign a single SI to a number of SUs.

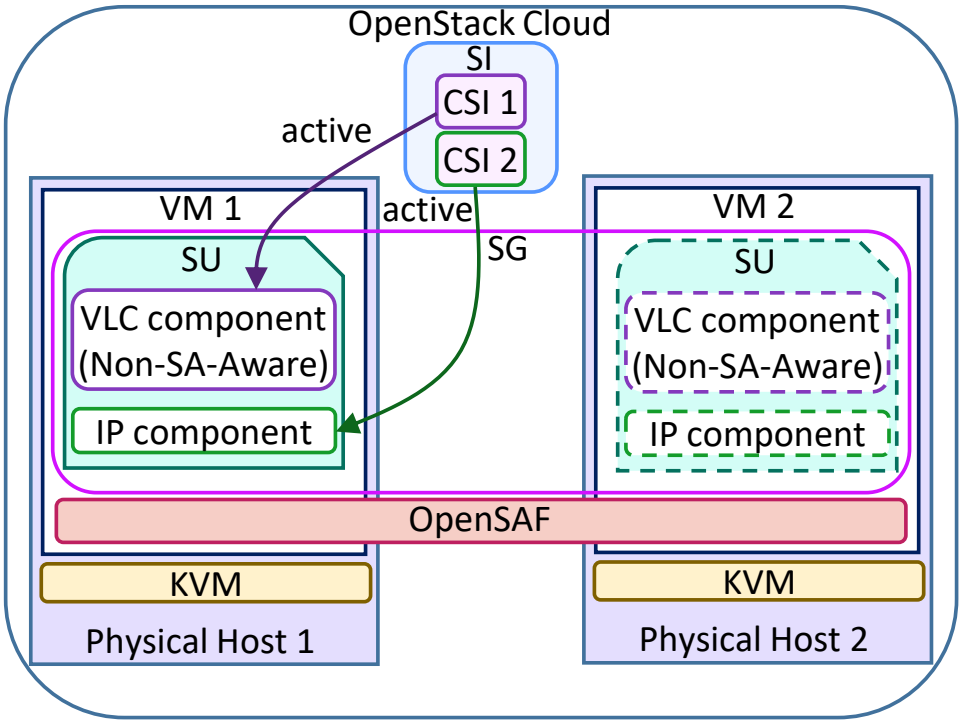

Figure 3-11. The architecture for availability experiments with OpenSAF (stateless VLC).

We considered the following failure scenarios of VLC process failure, VM failure, and physical host failure, corresponding to VLC container failure, pod process failure and node failure, respectively. In the experiments with OpenSAF, a No-redundancy redundancy model with two VLC components are considered. One component is running and providing service and the other one is a spare to be instantiated and take over in case of failure of the active. The results of the experiments with OpenSAF and the comparison with Kubernetes are shown in Table 3-6.

Table 3-6. Experiments with OpenSAF (Non-SA-Aware VLC).

failure trigger (unit: seconds)	reaction time	repair time	recovery time	outage time
VLC Process Failure	0.650	-	0.145	0.795
VM Failure	3.233	-	0.123	3.351
Physical Host Failure	3.229	-	0.118	3.346

3.3 Analysis and Discussion

In this section, we analyze the results of the availability experiments of Section 3.2 in order to answer the research questions we asked earlier and also discuss the availability challenges for stateful microservice based applications.

3.3.1 Availability of Stateless Applications

We analyze the results of the availability experiments of Section 3.2 separately for each failure scenario. First, we analyze the results of the scenarios where service outage is due to external execution failures and later compare with the results of the experiments where the service outage is due to administrative failure events.

Analysis of Service Outage due to Application Container Failure Scenario: In this failure scenario, after killing the application container, the service becomes unavailable. However, since Kubernetes has not detected the failure yet, the IP address of the failed pod stays in the endpoints list. The reaction time is when Kubernetes detects the failure and removes the pod's IP from the endpoints list. As it is observed in Table 3-1 and Table 3-3, for the architectures of Figure 3-9 and Figure 3-10, the measured reaction times are close (0.716 and 0.579 seconds). The total service outage, however, is different. Service outage for the architecture

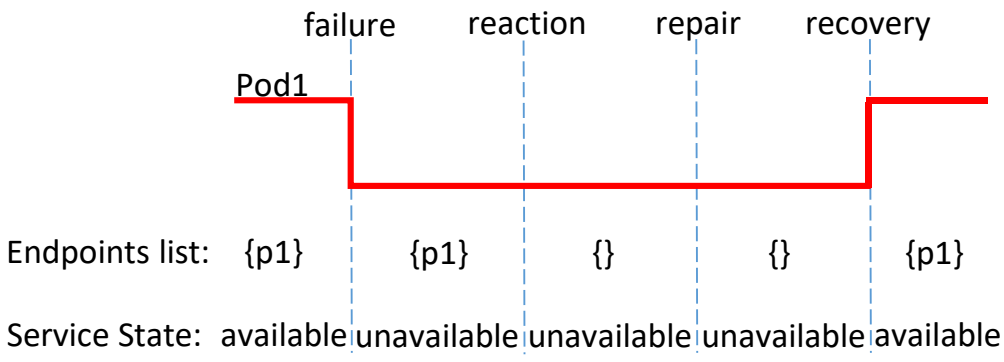

Figure 3-12. Analysis of experiments with Kubernetes under the default configuration and No-Redundancy redundancy model – evaluating the repair actions.

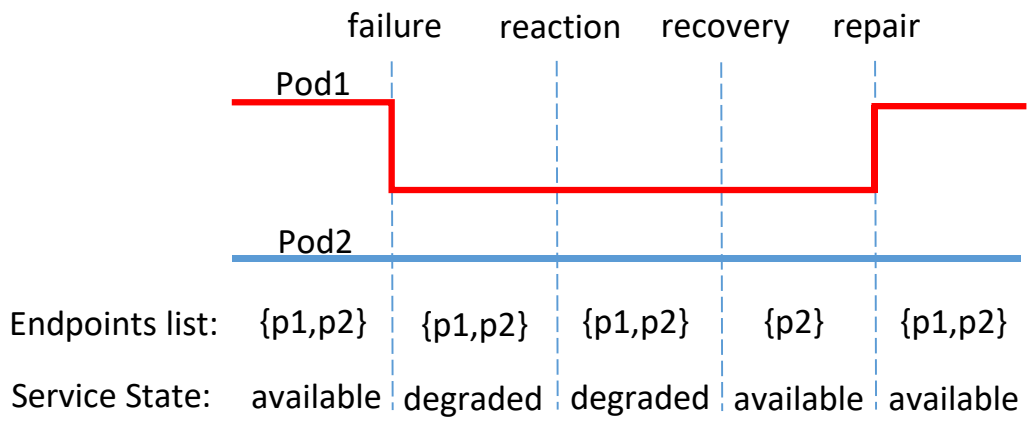

Figure 3-13. Analysis of experiments with Kubernetes under the default configuration and N-Way Active Redundancy model – evaluating the impact of redundancy.

with No-Redundancy redundancy model was measured 1.766 seconds while for the architecture with N-Way Active redundancy model was only measured 0.579 seconds. Because for the former, we rely on the failed pod to be repaired in order to have service recovery (Figure 3-12). However, for the latter, the service is recovered by only removing the unhealthy Pod1 from the endpoints list (Figure 3-13). This is because another healthy pod is still on the endpoints list and ready to serve the requests.

In Figure 3-14, we compare the results of the experiments in Sub-section 3.2.2.1 with those of the experiments with OpenSAF (Sub-section 3.2.2.4) as a proven reference for availability management. As it is observed in Table 3-1 and Table 3-6, the measured service outage for the experiments with Kubernetes with No-Redundancy redundancy model is higher than that of the experiments with OpenSAF (1.766 and 0.795 seconds). The recovery time of the experiments with OpenSAF is lower because there is a spare which is ready to be instantiated when a failure happens.

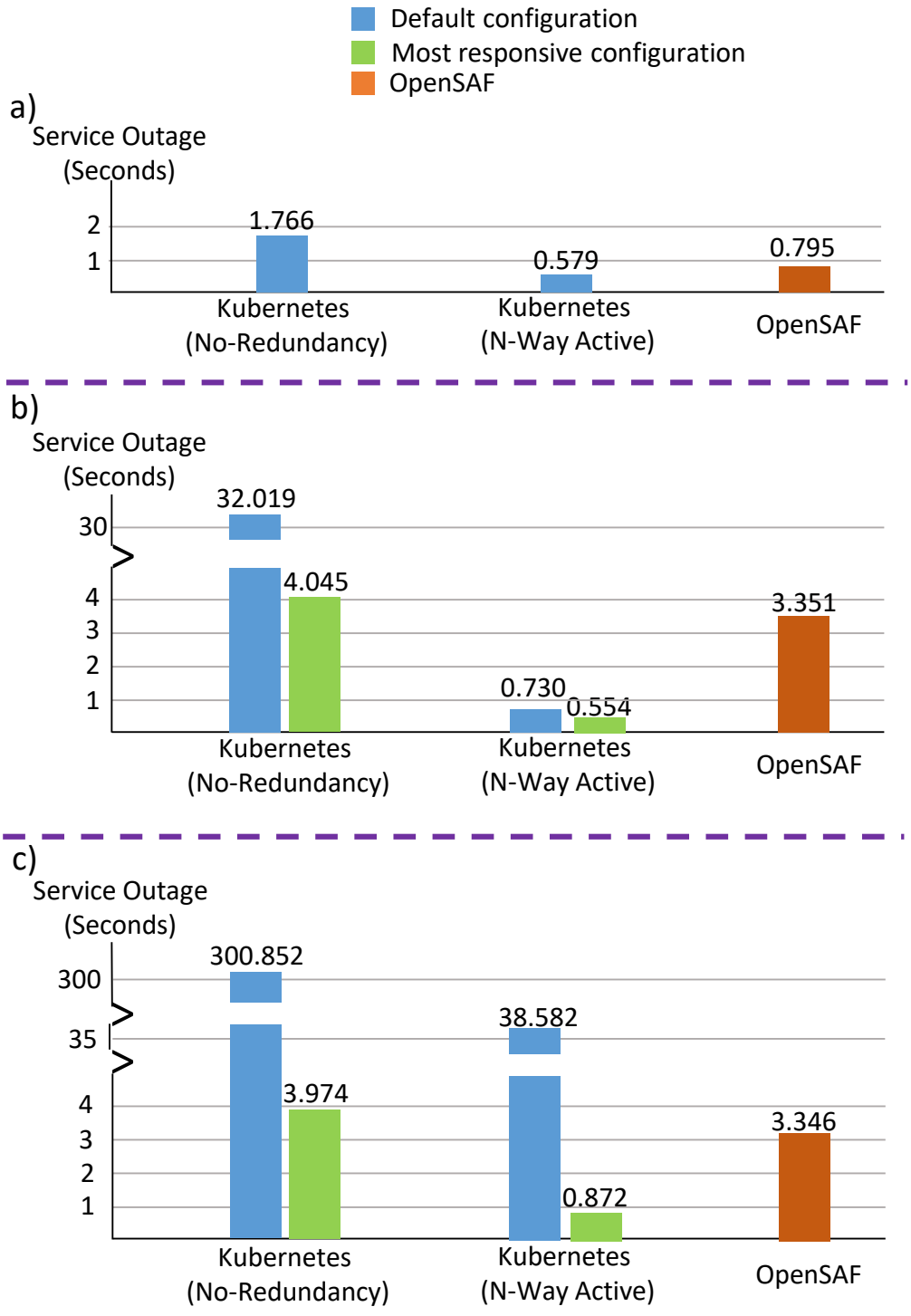

Figure 3-14. Comparing Kubernetes and OpenSAF from availability perspective for stateless applications. a) VLC container failure scenario, b) Pod container failure scenario, c) Node failure scenario.

Analysis of Service Outage due to Pod Process Failure: In this failure scenario, the service becomes unavailable when the pod process fails. In Table 3-1 and Table 3-3, we observe that the reaction time for the architectures of Figure 3-9 and Figure 3-10 are relatively close while their total service outage significantly differ (32.019 and 0.730 seconds). The reason for this difference is the repair time which takes 30 seconds on average and in the architecture with No-Redundancy redundancy model, recovery depends on the repair of the failed pod. In the architecture of Figure 3-10, however, recovery happens shortly after reaction time when Kubernetes removes the IP of the failed pod from the endpoints list. The reason for the long repair time is that when the pod process fails, a graceful termination signal is sent to the VLC container and Docker waits 30 seconds before terminating it forcefully and the repair process will not start unless the VLC container is terminated.

In Sub-section 3.2.2.3, we reconfigured the graceful termination period of the pod to decrease the aforementioned long repair time. The results of these experiments are presented in Table 3-4. As it was expected, Table 3-4 shows a significant decrease in repair time which affects the service outage of experiments done with No-Redundancy redundancy model. The service outage of experiments with the N-Way Active redundancy model has not changed, as the repair time does not play a role in the service outage in this case. We observed that with the new configuration when the pod process crashes, the time Docker gives to the application container before forcefully killing it is reduced to 2 seconds. However, this drastic change of the graceful termination period might cause unnecessary pod restarts when there are false-positive reports about the application container failure.

For the architecture of Figure 3-9, we can compare the results of the pod failure scenario with the results of the experiments with OpenSAF where service outage is due to VM failure. We observe in Figure 3-14 that with the default graceful termination period, the outage time

with Kubernetes is significantly higher compared to that of the experiments with OpenSAF. However, the service outage with Kubernetes and OpenSAF became comparable by reconfiguring the graceful termination period to its lowest value.

Analysis of Service Outage due to Node Failure Scenario: In this failure scenario, as it is observed in Table 3-1 and Table 3-3, with the default configuration of Kubernetes, it takes between 30 to 40 seconds for Kubernetes to consider the node as lost and remove the IP of its pods from the endpoints list and the repair time also takes around 260 seconds. While the service outage for the architecture with No-Redundancy redundancy model was measured 300.852 seconds, this metric was measured 38.582 seconds for the architecture with N-Way Active redundancy model. Because as mentioned before, for the architecture of Figure 3-10, recovery does not depend on the repair of the failed pod.

In Sub-section 3.2.2.3, we reconfigured the node failure handling parameters to their most responsive configuration. Table 3-1, Table 3-3, and Table 3-5 show that reconfiguring Kubernetes reduced the outage time from 300.852 seconds to 3.974 seconds for the architecture of Figure 3-9 and from 38.582 seconds to 0.897 seconds for the architecture of Figure 3-10.

Moreover, in comparison with OpenSAF (Figure 3-14), we observe that in the cases of No-Redundancy redundancy model, the OpenSAF solution shows a lower outage time. Moreover, although the N-Way Active redundancy model should render a higher level of availability compared to the No-Redundancy redundancy [45], the outage time for the node failure scenario of Kubernetes with N-Way Active is still significantly higher than for OpenSAF with the No-Redundancy redundancy model. The reason for the long outage time with Kubernetes is the default configuration of Kubernetes that leads to late reaction time. However, with the changed configuration of Kubernetes, the outage times in Kubernetes experiments with No-Redundancy architecture are comparable to those of the OpenSAF solution.

Administrative Failure Events vs Execution Failure Events: Now we compare the results of external execution failure scenarios with those of the administrative failure scenarios. In the administrative pod termination scenario, the reaction time is 0.041 seconds which is significantly lower than the 0.496 seconds of the externally triggered pod process failure. The reason is that in the former, the termination is triggered from inside of Kubernetes, which then reacts according to the termination procedure, while in the latter it is up to the Kubelet's health check to detect first that the pod is no longer present and this depends on how close to the next health check the failure happens.

An important observation of these experiments is shown in Figure 3-15. Although the pod process is failed forcefully (Figure 3-15 (b)), the orphaned application container of the pod

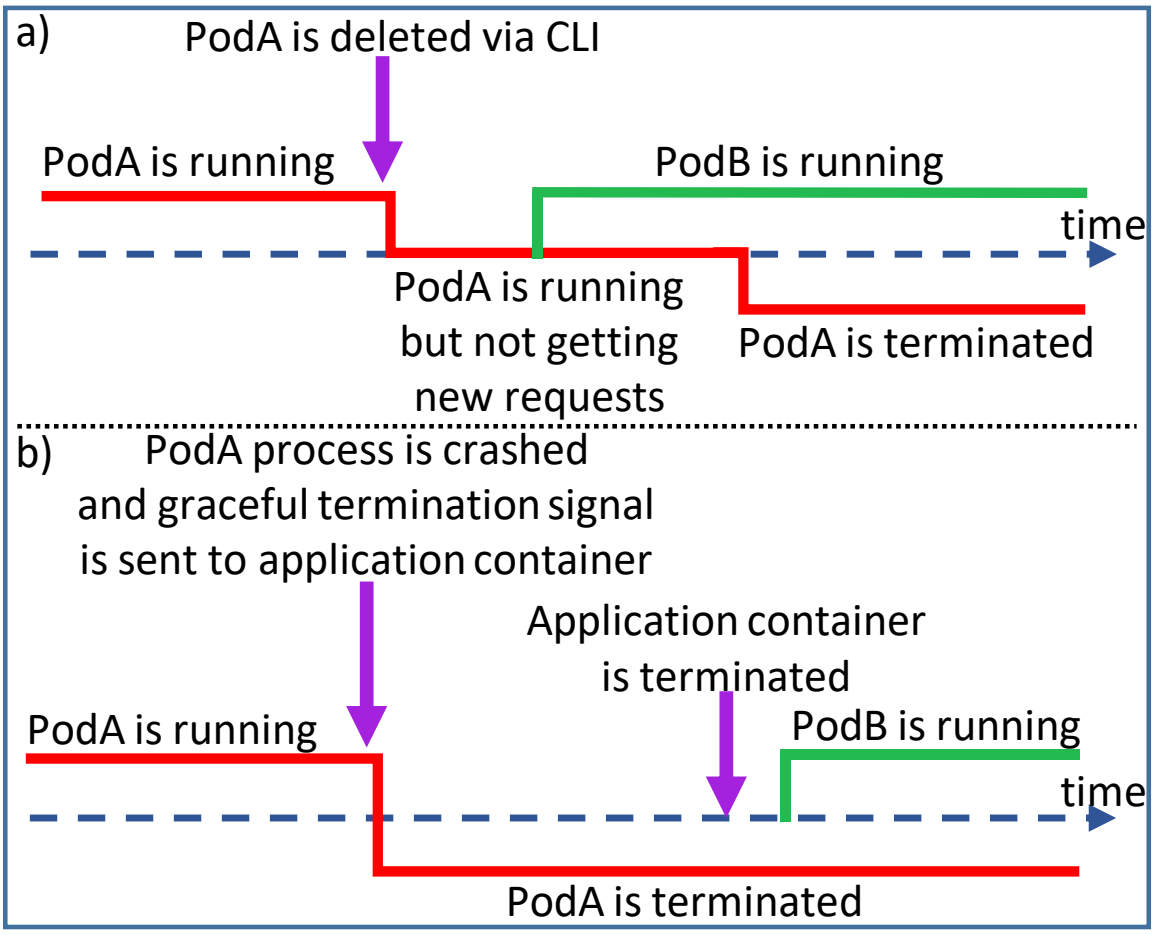

Figure 3-15. Analysis of pod failure scenarios. (a) Administrative pod termination. (b) Pod process failure.

receives a graceful termination signal. Thus, the pod process failure is detected by the Kubelet, which waits for Docker and will not start the repair process before it makes sure that the application container of the pod is terminated as well. This means graceful termination of the application container whose duration depends on Docker's configuration, impacts and delays the service recovery time. However, this may also allow for fault propagation should the pod process fail due to real fault or bug. Fault isolation principles would require immediate forceful cleanup of the application containers once their pod's process failure is detected. This grace period is the reason why the repair time for the pod process failure scenario is 32.570 seconds. This is significantly longer than that of the administrative pod termination scenario which is 0.982 seconds. In the latter (Figure 3-15 (a)), Kubernetes performs the graceful termination and repair procedures in parallel. The ordering is guaranteed for certain actions of these procedures. For example, the removal of the terminating pod from the endpoints list precedes the start of the repair procedure, and the completion of pod termination follows the addition of the new pod. This parallelization is possible due to the assumption that there is no fault in the system. This emphasizes the point made earlier about the correct simulation of failures with respect to availability metrics. As it is observed, the administrative pod termination scenario reports an outage time of 1.588 seconds while for the pod process failure scenario it is 32.019 seconds.

For the node failure scenarios, we observed similar differences in all measured availability metrics. For the administrative node deletion scenario, since the failure is triggered from inside of Kubernetes, the reaction time is 0.031 and it is significantly faster than the reaction time of externally triggered node failure which is 38.187 seconds. As explained before and shown in Figure 3-16 (b), it depends on the period of the Kubelet's status update (default 10 seconds) and the allowed number of misses (default 4 seconds).

Another important observation of the administrative node deletion scenario is shown in Figure 3-16 (a). Although the failure is triggered from inside of Kubernetes, the new pod is started after the old one is terminated. It was expected to behave similarly to the administrative pod termination shown in Figure 3-15 (a) where the new pod is started before the old one is terminated.

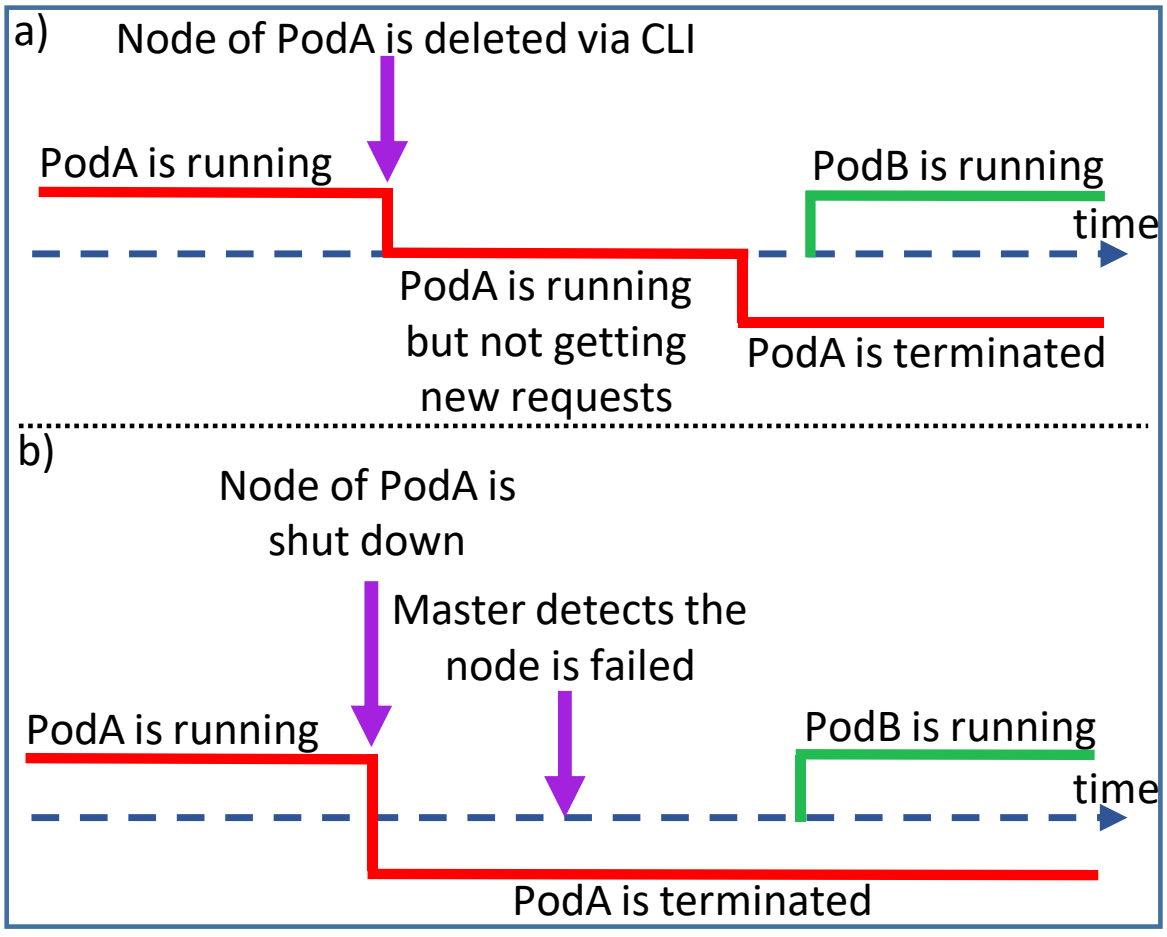

Figure 3-16. Analysis of node failure scenarios. (a) Administrative node termination. (b) Externally triggered node failure.

3.3.2 Challenges of Managing the Availability of Stateful Applications

As mentioned before, to manage the availability of applications, Kubernetes provides healing for its managed microservices [46] that is restarting the failed containers and replacing or rescheduling them when their hosts fail. Although these repair actions can improve the availability of the applications deployed with Kubernetes, redundancy remains the most important feature to enable high availability (HA).

Kubernetes enables replication of the pods with the aim of improving applications' availability. Stateless pods can be easily replicated as they can be deployed as interchangeable instances. However, the same is not true for stateful pods. Deploying a replicated set of stateful pods requires coordination of the different replicas to keep them synchronized and the "state" aspect makes orchestration more complex than what the initial Kubernetes features and controllers were built for.

For example, if the application is deployed by a StatefulSet controller as shown in Figure 3-2, if one pod fails, other pods cannot resume the service instead of the failed pod. One reason is that the state data for each pod are stored separately and other pods do not have access to them so they cannot resume the service. The other reason is that pods are isolated and are not aware of each other's failure and therefore cannot resume the service instead of the failed pod. Therefore, we can only rely on the failed pod to be restarted with the same identity so it can restore the state that was stored before failure from its own PV. This means that the service can be recovered, but the clients need to wait for the failed pod to be restarted, which may be too slow for some applications or impossible for some failure scenarios. For example, with the architecture in Figure 3-2, if the service outage is due to node shutdown, the pod will not be restarted and the service will not be recovered unless the node rejoins the cluster.

Moreover, as shown in Figure 3-3, when a stateful microservice based application is deployed by a Deployment controller, all microservice instances have access to the same state data. Because it is not possible to define separate PVs for each pod of a Deployment controller. However, if a pod fails, other pods will not be able to resume the service that was provided by the failed pod. Because although every pod has access to the state data, they do not know about the failure and the need for the service to be recovered. Moreover, they are not aware of the location where the failed pod's state data is stored. Therefore, they cannot recover the service that was provided by the failed pod. Since the identity of a restarted pod also changes, we cannot rely on the restart procedure for recovering the stored service state.

3.4 Conclusion

In this chapter, we presented and compared architectures for deploying stateless and stateful microservice based applications in Kubernetes clusters hosted in public and private clouds. Through our investigations, we learned that although it is not stated in Kubernetes' documentation [10], Kubernetes is more tailored for public clouds than it is for private clouds. We also conducted experiments in a private cloud environment, considering different failure scenarios, configurations, and redundancy models to evaluate Kubernetes from the perspective of availability it can provide for its applications.

In our experiments, we found that the failure scenarios due to external execution failure events show significantly longer outage times compared to failure scenarios due to internal administrative operations that are more commonly used to demonstrate Kubernetes' support for availability. We also found that the repair actions of Kubernetes are not sufficient for providing availability, especially high availability. For instance, the default configuration of Kubernetes results in a significant outage in the case of node failure. The outage time for this scenario was measured around 5 minutes, which is equivalent to the amount of downtime allowed in a one-year period for a highly available system. Kubernetes can be reconfigured to avoid this significant outage and under its most responsive configuration, outage times in Kubernetes experiments are comparable to those of OpenSAF, a proven solution for availability management. However, the impact of this reconfiguration on network overhead and false-positive node failure reports should be investigated. Moreover, the results of our experiments are compared with the experiments with OpenSAF where the VLC component is not modified. However, if the VLC component is modified and implement the AMF API, higher service availability will be achieved.

We acknowledge that there are some threats to the validity of our results. For example, all experiments were conducted in a small cluster consisting of only a master and two worker nodes. Kubernetes may behave differently in larger clusters which may impact the availability measurements presented in our experiments. Also, the availability measurements may also vary depending on the application's complexity and the collocated applications managed by Kubernetes. In our experiments, we considered a simple case of only one microservice. We understand that these factors may impact the results of our study. The mapping of the metrics to the concrete events is the biggest threat and requires more investigation as one can map them differently, in which case all the measurements could be different. However, we believe that even with a different mapping what would change is the split between reaction and repair times and reaction and recovery times, thus, resulting still in the same outage time. We may observe a decrease in the reaction time which adds to the recovery time, or inversely but the total outage time would be the same since it represents the duration in which the service was not available.

For stateless applications, we observed that adding redundancy can significantly decrease the downtime since the service is recovered as soon as Kubernetes detects the failure and it does not depend on the repair of the faulty unit. For the stateful applications, however, adding redundancy cannot improve the availability of the application. Because if one pod fails, the redundant pod neither knows about the failure nor has the state data and therefore cannot take over and resume the service that was previously provided by the failed pod. Therefore, we rely on the failed pod to be repaired so it can resume the service. In chapter 4, we will propose a solution to address the challenges that Kubernetes has in providing availability for stateful microservice based applications.

The contents provided in this chapter are published in [47] and [48].

Chapter 4

A State Controller to Manage the Availability of Stateful Microservice Based Applications

In Section 3.3, we explained the issues that Kubernetes has in managing the availability of stateful applications. In this chapter, we address these issues and propose a solution which is integrated with Kubernetes and improves the availability of stateful microservice based applications. This solution, which is explained in Section 4.1, consists of a State Controller that allows for the automatic service redirection to the healthy pods through the usage and management of secondary pod labels. We have implemented a prototype for the State Controller and conducted availability experiments to evaluate the availability provided by our solution and compare with the availability of the architectures without our solution. Moreover, in Section 4.2, we enrich the State Controller so it can handle the cases where the application is scaled in or scaled out and provide availability for the stateful microservice based applications whose number of microservice instances change. We also implemented a prototype of the State Controller enriched with elasticity and conducted experiments to evaluate the modified State Controller in terms of scaling overhead and availability.

4.1 A State Controller for Kubernetes

We explained the challenges of managing the availability of stateful microservice based applications with Kubernetes in Section 3.3. The main problem is that when a pod instance fails, other pods do not know about this failure and cannot recover the service that was provided by the failed pod. One solution is to have a standby pod that would take over and resume the service when the active pod fails. Moreover, there needs to be a third party that notifies the standby pod about the active pod's failure so it can resume the service. However in Kubernetes, the concept of standby does not exist and once a pod is deployed and added to the endpoints list of the service that exposes the application, it will be active and will serve once it receives a request.

4.1.1 Managing Availability with the State Controller

To address the issues mentioned at the beginning of this section, we propose a solution that integrates the concept of high availability states, i.e., active and standby, with Kubernetes to improve the availability of stateful microservice based applications. In this solution, a component named the State Controller is added to Kubernetes per service. The State Controller communicates with the API server and assigns a secondary label (HAState label) with the value of active or standby to the pods.

To expose the application, a service which we call the application service should be created that uses the HAState label and only targets the pod whose HAState label has the active value. In addition to the service exposing the application, another service called the state replication service should be created which only targets the pod whose HAState label has the standby value. Through the state replication service, the checkpointing process of the active

pod which has the IP address of the state replication service sends the state data to the standby pod every time it checkpoints its state data to its PV.

What follows are the steps the State Controller takes to manage the availability of stateful applications (Figure 4-1):

A. Assigns HAStates to pods (active and standby)
    1) The active pod becomes the endpoint of the service exposing the application (application service)
    2) The standby pod becomes the endpoint of the state replication service
B. Monitors for the events related to pods and if it identifies a failure, it will take action accordingly
    1) If the failed pod had the active HAState, it assigns standby HAState to the failed pod and active HAState to the standby pod. The new active pod becomes the endpoint of the application service and restores the last state from its storage area in the PV and resumes the service.
    2) If the failed pod had the standby HAState, it ensures that the failed pod is assigned the standby HAState after it is repaired.

This solution enables availability management of stateful microservice based applications deployed by a Deployment controller and speeds up the service recovery for the ones that are deployed by a StatefulSet controller. Because service recovery no longer depends on the repair of the failed pod.

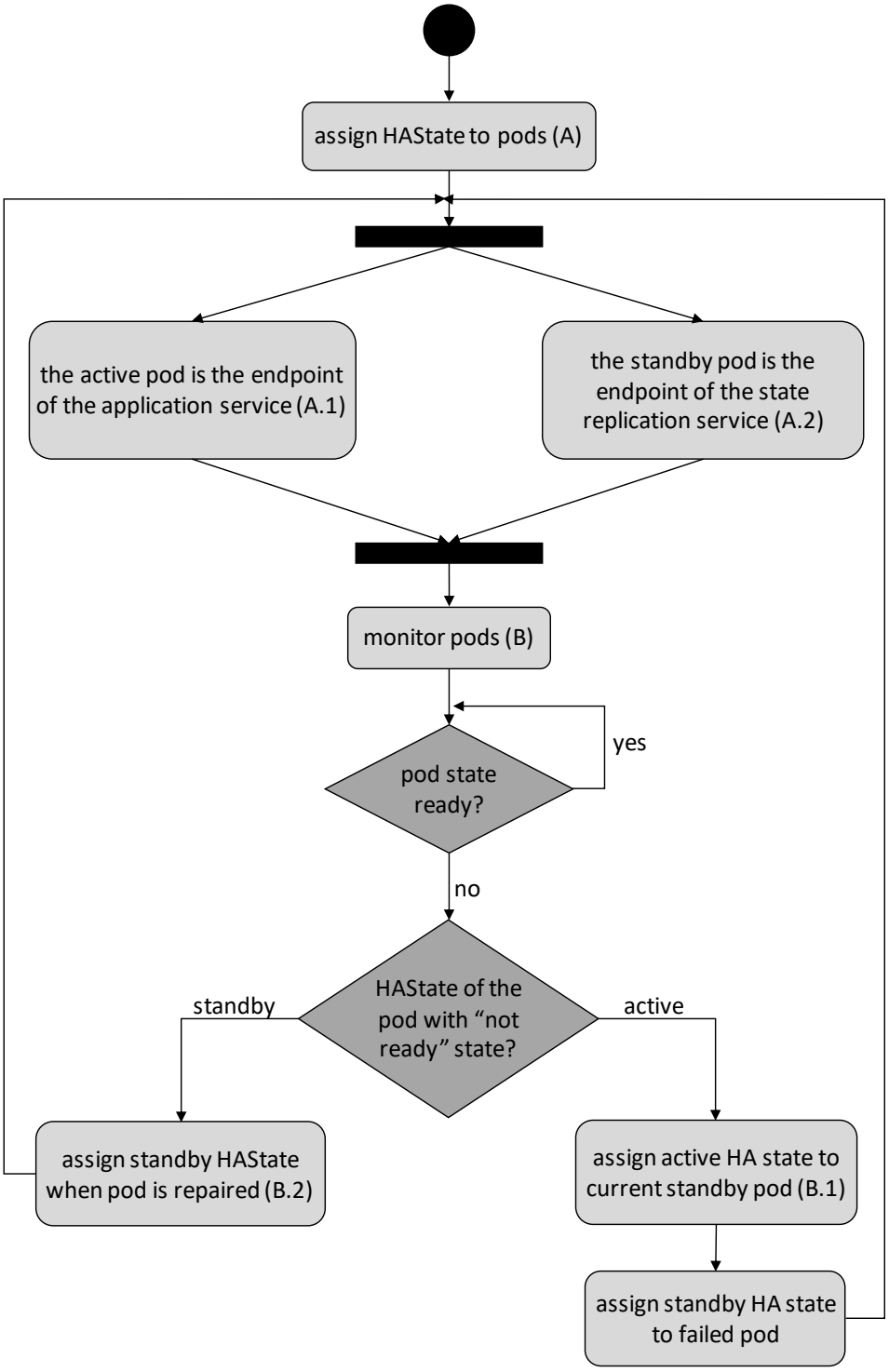

Figure 4-1. The behavior of the State Controller.

As mentioned above, one role of the State Controller is to assign an active label to one pod and standby label to another pod after they are deployed (Figure 4-2). In Figure 4-2, "x" can be either active or standby. It is important to note that pods are not aware of their HAState

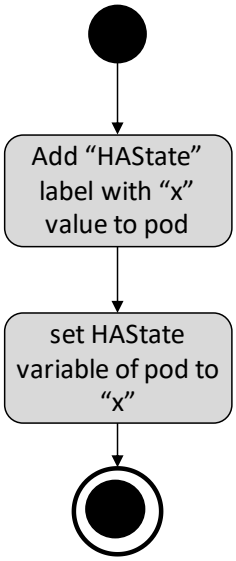

Figure 4-2. Setting the HAState label and HAState variable for pods (Step A).

labels assigned by the State Controller. Therefore, to make the pods aware of their HA states, the State Controller manipulates an environment variable in each pods’ environment called the HAState variable which is also shown in Figure 4-2.

As the State Controller makes changes in the HAState variable of the pod, there needs to be an agent inside each pod that coordinates the pod’s actions with respect to these changes. Therefore, a script is included in the container image of pods that is executed as the entrypoint process of the pod and runs in the background once the pod is deployed. The entrypoint process has two tasks which are explained in Figure 4-3 and Figure 4-4.

A. The first task of the entrypoint process explained in Figure 4-3 is to monitor the HAState variable of the pod and make decisions accordingly:

    1) Before the HAState variable is set by the State Controller, its value is “not set”. The entrypoint process keeps checking until the HAState variable is set to active or standby. To check for changes made to the HAState variable, the entrypoint

process keeps the current HA state of the pod and periodically compares the HAState variable and the current HA state.

2) If the HAState variable is active, and it is changed by the State Controller to standby, the entrypoint process initiates a self-cleanup process that terminates all processes running in the pod's environment, except the entrypoint process itself. The reason for the self-cleanup is to ensure that if an active is changed into standby by mistake, it keeps serving and changing the data.

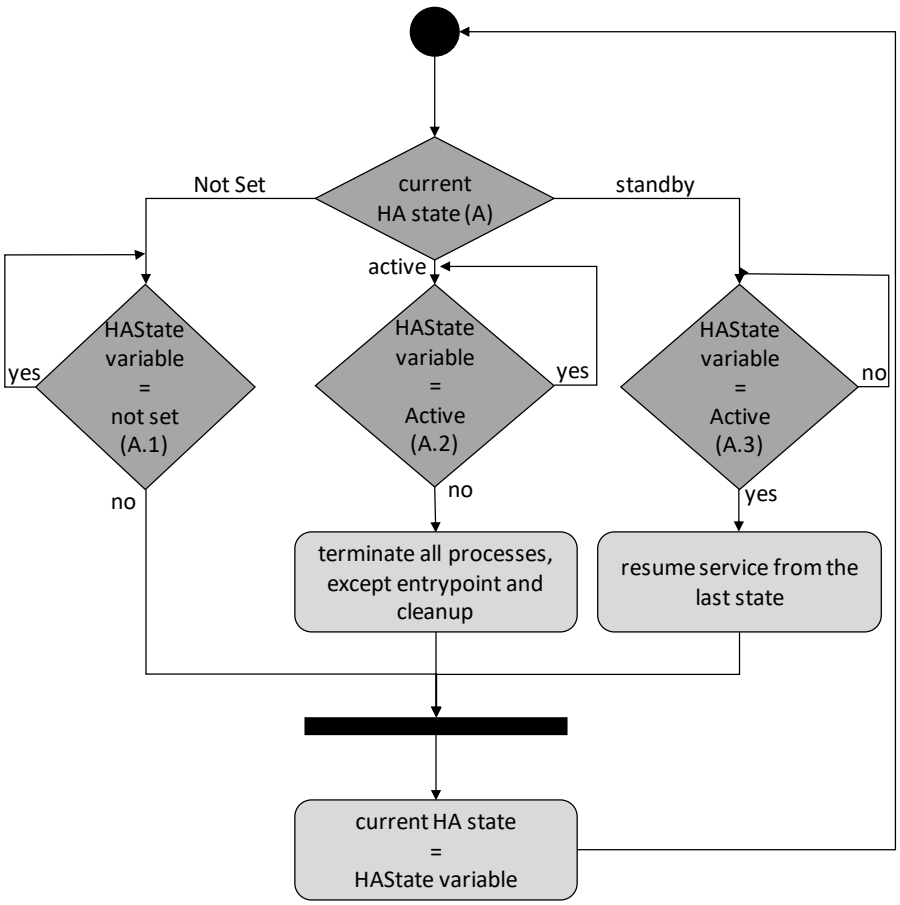

Figure 4-3. Decision making of the entrypoint process based on the HAState variable (Step A).

3) When the HAState variable is standby, and it is changed by the State Controller to active, it restores the last state stored by the failed pod from the PV and resumes the service and starts checkpointing the state in its own storage area in the PV.

B. The other task of the entrypoint process which is explained in Figure 4-4 is to run a watch loop to enable initiating a self-cleanup process when the pod is no longer controlled by Kubernetes (e.g., a network partition has happened between the pod and the master) to avoid data loss or data inconsistency. In this watch loop, the master is periodically pinged and if it is unreachable for three tries, a self-cleanup procedure will be initiated that will terminate all processes except the entrypoint process itself. In the self-cleanup procedure, all processes except for the entrypoint process are forcefully terminated.

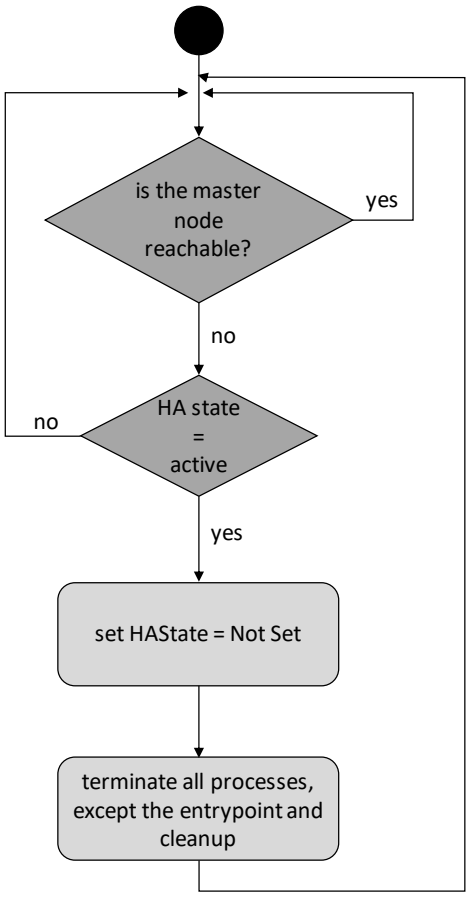

Figure 4-4. Self-cleanup watch loop of the entrypoint process (Step B).

The proposed State Controller has been implemented as a proof of concept. The State Controller has been developed using the Go programming language [49]. For the implementation, we used the client-go library [50] of Kubernetes which consumes the REST interface exposed by Kubernetes API server in order to access and manipulate objects deployed with Kubernetes (pods, services, and etc.). In our implementation, we retrieve the configuration of the Kubernetes cluster and create a client that communicates with the API server and monitors the events related to pod objects exposed by Kubernetes and stores them in a message queue. Our proposed State Controller reads this message queue and performs the tasks explained at the beginning of this section.

Our solution can be used to manage the availability of applications deployed with StatefulSet controllers and with Deployment controllers. In the following sub-sections, we bring the architectures where we integrate the State Controller with StatefulSet controllers as well as the ones deployed with Deployment controllers and evaluate them. In any case, for recovering the failed pods the State Controller relies on the Deployment or StatefulSet controller without any modifications.

4.1.2 Integrating the State Controller with Kubernetes

In this sub-section, we present the architectures where the proposed State Controller is integrated with Kubernetes.

4.1.2.1 *Integrating the State Controller with StatefulSet Controllers*

Figure 4-5 shows the architecture where the proposed State Controller is integrated with Kubernetes to manage the availability of a stateful application deployed with a StatefulSet controller (named "VoD"). In this architecture, the StatefulSet controller is deployed and creates two pod replicas. Each pod has a separate PV where it can store its state data. In this architecture, two services should be created by the user:

i. The application service which exposes the active pod to the clients. This service only targets the pods that have both labels of "app: VoD" and "HAState: Active".

ii. The state replication service which has a static IP address that does not change and is known to the active pod as an endpoint where it should replicate its state data to. This

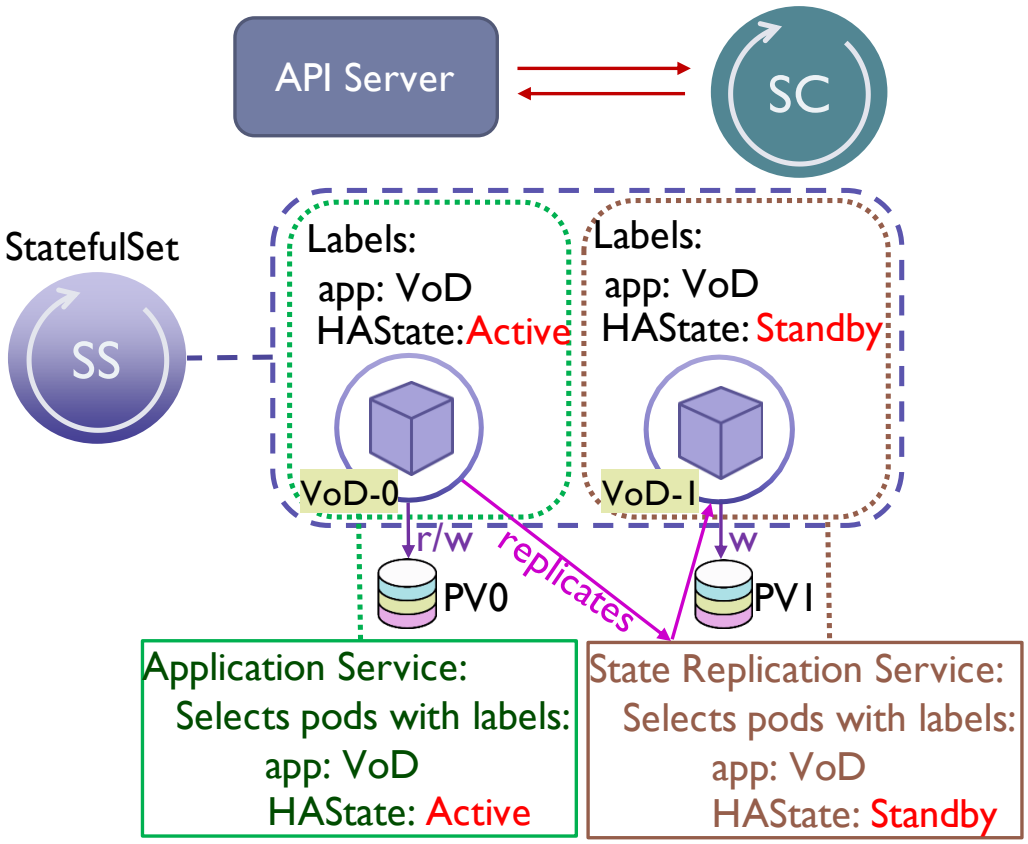

Figure 4-5. Integrating the State Controller with StatefulSet controllers.

service only targets the pods who have both labels of “app: VoD” and “HAState: Standby”. In the checkpointing process included in the container image of the pods, when a pod is active, it periodically stores its state in its own PV and also replicates it to the IP address of the state replication service. The state replication service receives the state data and sends it to the standby pod through an HTTP request. The standby pod that is listening will store the state data in its own PV when it receives it. Through the state replication service, the active pod does not need to keep track of the changes in the location of the standby pod.

As mentioned before, the State Controller monitors the events related to pods and if a failure happens to the active pod, it changes the HAState label value of the standby pod from standby into active, making it the new endpoint to the application service. Consequently, the entrypoint process of the new active pod will initiate the service resume process where the last stored state is retrieved from the associated PV and the service is resumed. The new active pod has the state data of the failed active pod because the failed active pod had been replicating its state to the standby pod through the state replication service. The HAState label of the failed pod is also changed from active to standby once it is repaired.

As mentioned before, StatefulSet controllers do not recreate pods if the pods’ hosts fail and it is not possible to recover the service unless the host is repaired and rejoins the cluster. However, integrating the State Controller with StatefulSet controllers enables service recovery for this failure scenario. Because regardless of the cause of a failure, the State Controller changes the HAState label of the standby pod into active and because of the change in the label, the application service is redirected to the new active pod. It is important to mention that although the service is recovered because of changing HAState labels by the State Controller, the failed pod will still not be repaired by the StatefulSet controller unless the node becomes responsive again. Meaning that with this architecture, it is only possible to recover from only one

node failure. Because after each node failure, one pod will be lost, unless its node becomes responsive again.

If the node failure was due to a network partition, the entrypoint process of the active pod whose host had left the cluster would detect that the master is out of reach. Therefore, it would automatically set its HAState variable to “not set” and terminate all running processes except for the entrypoint process itself. This way, we ensure that the former active pod will not keep serving the clients.

Since Deployment controllers do not have limitations in recovering from node failures, we also bring an architecture where we integrate the State Controller with a Deployment controller in the next sub-section.

4.1.2.2 *Integrating the State Controller with Deployment Controllers*

Figure 4-6 shows the architecture where the State Controller is integrated with a Deployment controller to manage the availability of a stateful application. Similar to the architecture of Figure 4-5, two pod replicas are deployed. We have the application service to expose the application and the state replication service for the active pod to replicate its state to the standby pod. The difference is that with Deployment controllers, the same PV is shared between all pods. However, we create a separate storage area for each pod to distinguish between the state data of each pod that is stored for each client.

The steps that the State Controller takes for managing the availability of the applications deployed with Deployment controllers are the same as described in the previous subsection. The difference is in how the Deployment controller behaves when a node failure is detected. As Deployment controllers are primarily used for stateless applications, they do not consider the risk of data inconsistency in case of a network partition and recreate pods when their hosts become unresponsive. For example, if the node hosting PodA in Figure 4-6 becomes unresponsive, whether it is due to a network partition or a system reboot, the Deployment controller will delete the failed pod from the list of pods and recreate another pod on a healthy node after the pod eviction timeout which is defined in the configuration. Therefore, with this architecture, the number of pod replicas do not change with node failures.

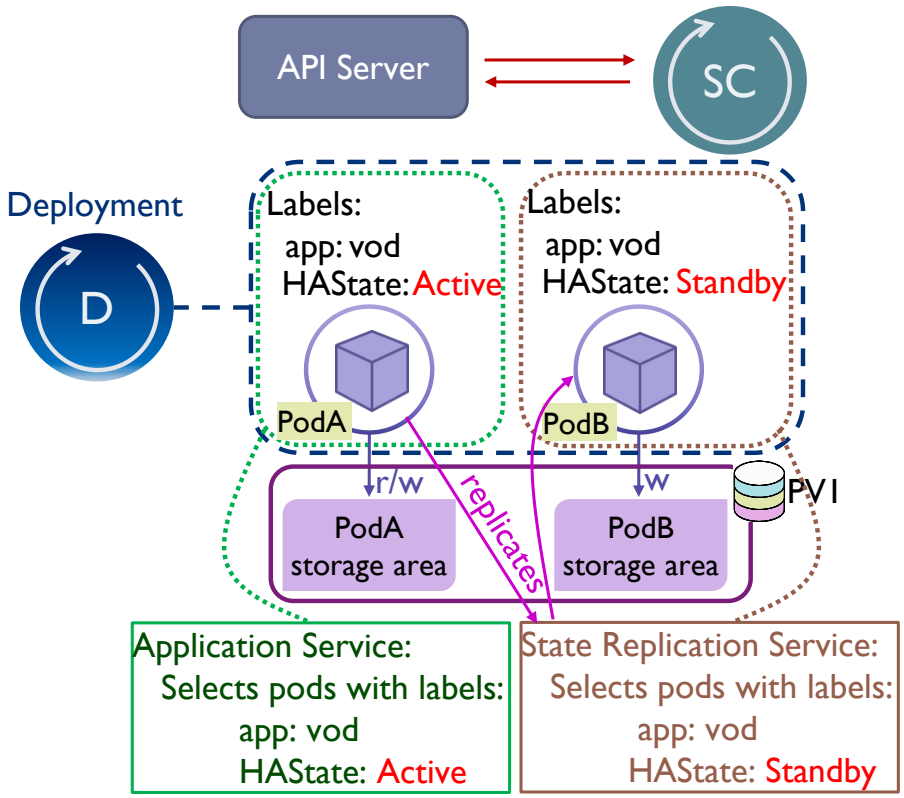

Figure 4-6. Integrating the State Controller with Deployment controllers.

4.1.3 Evaluating the Achievable Service Availability by Integrating the State Controller with Kubernetes

In Section 3.3, we explained the issues that Kubernetes has in managing the availability of stateful microservice based applications and in Section 4.1, we proposed a solution in order to enrich Kubernetes with a State Controller so it can achieve a higher level of availability through managing redundant stateful microservices. In this sub-section, we evaluate the achievable service availability with the architectures presented in Section 4.1 where we integrate the State Controller with a StatefulSet controller (Figure 4-5) and with a Deployment controller (Figure 4-6). We evaluate these architectures by addressing the following research questions (RQ):

RQ5: What is the level of availability that Kubernetes can provide for stateful microservices solely through its repair actions?

RQ6: What is the impact of enriching Kubernetes with the proposed State Controller for state management on the availability of stateful microservice based applications?

RQ7: What is the availability achievable with the State Controller for stateful microservice based applications under the most responsive configuration of Kubernetes?

RQ8: How does the availability achievable with the State Controller for stateful microservice based applications compare to non-Kubernetes based solutions?

To address these research questions, we conducted a set of availability experiments under the failure scenarios introduced in Sub-section 3.2.1.2 and measured the availability metrics defined in Sub-section 3.2.1.1.

In these experiments, we have a Kubernetes cluster composed of four VMs running on OpenStack cloud. Ubuntu 16.04 is the OS running on all VMs. Kubernetes 1.12.1 runs on all VMs and the container engine is Docker 18.06. NTP is used for time synchronization between VMs. The application deployed is stateful Video on Demand (VoD) where each client can request a video to be streamed for them. The same pod template is used for all experiments that has one container image in which VLC is installed as the video streaming application. The container image also has the Apache HTTP server [51] hosting a webpage that allows the clients to request for a video stream. To ensure service continuity, the container image has a checkpointing process which checkpoints the elapsed time of the video to the location where its PV is mounted when the pod receives requests from clients to stream a video. The streaming position, which is the state data, is stored for each client separately. The state data is also sent to the IP address of the state replication service so it will be stored by the standby pod. The State Controller used in these experiments is the initial version which handles one active and one standby HA state assignment.

4.1.3.1 *Evaluating the Repair Actions with the Default Configuration of Kubernetes (RQ5)*

In Sub-section 3.1.2, we presented two possible architectures for deploying stateful microservice based applications with Kubernetes. One with StatefulSet controller (Figure 3-2) and the other with Deployment controller (Figure 3-3). In the latter, all microservice instances have access to the same state data. However, if a pod fails, other pods do not know about the failure, nor are aware of the location its data are stored. Therefore, they cannot recover the service that was provided by the failed pod. Since the identity of a restarted pod also changes in node failure scenarios, we cannot rely even on the restart procedure for recovering the stored service state. Therefore in this sub-section, we only evaluate the architecture where the application is deployed with a StatefulSet controller. Figure 4-7 shows the concrete architecture for the experiments where we evaluate Kubernetes in terms of the availability it provides for stateful applications only through its repair actions.

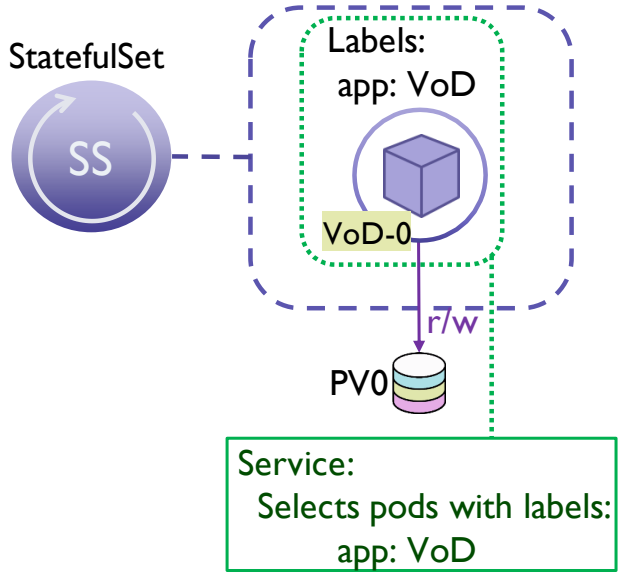

Figure 4-7. Concrete architecture for experimenting with Kubernetes - Stateful microservice based application with No-Redundancy redundancy model.

To answer this research question, we evaluate the availability metrics for each of the failure scenarios under the default configuration of Kubernetes through a set of availability experiments. The measurements of this set of experiments are shown in Table 4-1. What follows is the detailed explanation for each failure scenario in these experiments.

Service Outage due to Application Container Failure: In this scenario, the failure is simulated by killing the application container process from the OS. When the application container crashes, the Kubelet detects the crash and brings the pod to the "not ready" state where it will not receive new requests. At this time, that is the reaction time, the pod is removed from the endpoints list. Later, the Kubelet restarts the failed container and the pod is in the endpoints list of the service again and is ready to receive requests. This time marks the repair time. Since the application container was checkpointing its state to its PV before the failure, when the container is restarted, it can restore the last state from its PV and the video will start from the point of failure. This time marks the recovery time.

Service Outage due to Pod Process Failure: In this scenario, the pod process is killed from the OS which is detected and reported by the Kubelet. However, the Kubelet does not bring the pod to the "not ready" state and therefore the service is not interrupted. To repair the pod process, the Kubelet first terminates the application container which will cause service outage, and then starts the pod process and application container again. In this scenario, the failure event and reaction are considered at the same time which is when the application container is terminated. We consider the pod as repaired when the restart of both application container and pod process are finished. After restart, the application container restores the last stored state from its PV and the video will continue from this point, which marks the recovery time.

Service Outage due to Node Failure: In this scenario, node failure is simulated by shutting down the VM hosting the pod from OpenStack. As mentioned before, the Kubelet is responsible to report the status of the node to the master, and it is the node controller of the master who detects the failure of the node. When a node hosting a pod fails or is partitioned from the master, it stops sending status updates to the master and the master will mark the node as not ready after the fourth missed status update. This time is the reaction time. However, unlike Deployment controllers, StatefulSet controllers do not recreate pods if their hosts die, unless the node becomes responsive again. Therefore with this architecture, the pod will not be repaired in this scenario and the service cannot be recovered. We also simulated node failure by rebooting the VM hosting the pod using the Linux's reboot command so it becomes responsive again.

Table 4-1. Kubernetes with Default Configuration - Stateful VoD deployed with a StatefulSet controller.

failure trigger (unit: seconds)	reaction time	repair time	recovery time	outage time
Application Container Failure	0.679	1.029	1.480	2.159
Pod Process Failure	0	0.943	2.133	2.133
Node Shutdown	NA	NA	NA	NA
Node Reboot	37.127	126.400	127.380	164.507

4.1.3.2 *Evaluating the Impact of Enriching Kubernetes with the State Controller for State Management on the Availability of Stateful Microservice Based Applications (RQ6)*

To answer this research question, we conduct experiments to evaluate the achievable service availability when our proposed State Controller is integrated with StatefulSet controllers (Figure 4-5) as well as Deployment controllers (Figure 4-6). We evaluate the availability metrics for each of the failure scenarios under the default configuration of Kubernetes to later compare with the results of the experiments in Sub-section 4.1.3.1. The measurements of these experiments for the architectures of Figure 4-5 and Figure 4-6 are shown in Table 4-2 and Table 4-3, respectively. What follows is the detailed explanation for each failure scenario in these experiments.

Service Outage due to Application Container Process Failure: In this scenario for both architectures of Figure 4-5 and Figure 4-6, the failure happens when the application container process of the active pod is killed from the OS. When the active pod's applications container is killed, the Kubelet detects it and brings the active pod to the "not ready" state and removes it from the endpoints list of the application service which marks the reaction time. The State Controller reacts to the change made to the service state of the active pod and changes the HAState label and HAState variable of the standby pod to active. The entrypoint process of the new active pod detects the HAState variable's change and orders the pod to read the last stored state and resumes the service by executing the resume script present on the container image. Recovery time is when the new active pod has started the video stream from the last stored state. In the meantime, the failed pod is restarted by the Kubelet and gets the standby HAState label and HAState variable.

Service Outage due to Pod Process Failure: In this scenario, the pod process is killed from the OS and the Kubelet detects and reports that the pod process no longer exists but it does not bring the pod to the "not ready" state. The State Controller detects the event reported by the Kubelet and, since there is a healthy standby pod available, the State Controller initiates a failover to this standby pod by changing the HA state of the active pod to standby and terminating all its active processes. This results in a service outage and we consider this event as the failure event but also as the reaction to the failure. The reason that we consider this event also as the failure event is that prior to this event, although the pod process had crashed, the service was still available and based on our definition, the failure event is the start of service outage. Next, the State Controller assigns the active HA state to the healthy pod. The recovery time is when the new active pod starts streaming the video. As it was mentioned before, the repair time is when the restart of the pod with the failed pod process is completed.

Service Outage due to Node Failure: In this scenario for both architectures of Figure 4-5 and Figure 4-6, failure of the node hosting the active pod is simulated in two ways. One is by shutting down the node from OpenStack while the other is using Linux's reboot command on that node. After the node is considered as not ready (after four missed status updates in the default configuration), Kubernetes brings the active pod to the "not ready" state (reaction time). Therefore, the State Controller detects the failure and initiates the failover process. That is, the standby pod is assigned the active HAState label and variable and its entrypoint process will order the pod to resume the service by executing the resume script present on the container marking the recovery time. In the case of VM shutdown for the architecture shown in Figure 4-5, since the application is deployed by a StatefulSet controller, the failed pod will not be repaired. However, when the VM is rebooted, the StatefulSet controller will recreate the pod and therefore we have repair time. The repair time depends on how fast the node can reboot.

In both cases of VM shutdown and VM reboot for the architecture of Figure 4-6, since the application is deployed with a Deployment controller, the failed pod will be repaired. In all cases, if the pod is repaired, it will be assigned standby HAState.

Table 4-2. Kubernetes with Default Configuration - Stateful VoD deployed with a StatefulSet controller and the State Controller.

failure trigger (unit: seconds)	reaction time	repair time	recovery time	outage time
App Container Failure	0.739	1.052	0.661	1.400
Pod Process Failure	0	31.527	0.637	0.637
Node Shutdown	37.236	NA	0.710	37.946
Node Reboot	37.660	126.738	0.800	38.460

Table 4-3. Kubernetes with Default Configuration - Stateful VoD deployed with a Deployment Controller and the State Controller.

failure trigger (unit: seconds)	reaction time	repair time	recovery time	outage time
App Container Failure	0.549	1.027	0.656	1.205
Pod Process Failure	0	31.841	0.656	0.656
Node Shutdown	37.902	262.932	0.760	38.662
Node Reboot	36.128	123.974	0.827	36.955

4.1.3.3 *Evaluating the Availability Achievable with the State Controller for Stateful Microservice Based Applications under the Most Responsive Configuration of Kubernetes (RQ7)*

In the node failure scenarios, the default configuration of Kubernetes significantly delays both reaction and repair time. The configuration parameters are the frequency of posting the node status by the Kubelet to the master, the number of allowed missed status updates before marking a node as unresponsive, and pod eviction timeout. Therefore to answer RQ7, we conducted the experiments for the architectures of Figure 4-7, Figure 4-5, and Figure 4-6 under the most responsive configuration. Note that unlike what we discussed in Sub-section 3.2.2.3, the default graceful termination period for pods does not impact service outage in the newer versions of Kubernetes. Since in these experiments we used Kubernetes 1.12.1, we do not consider the case where we change the default graceful termination period parameter.

To have the most responsive configuration, we reconfigured the Kubelets of all nodes to post the status of the node to the master every one second. We also reconfigured the Controller Manager so the master checks the posted node statuses every second and allow one missed status update for each node. The pod eviction timeout is also set to one second. Since these parameters only affect responding to node failures, we only consider the node failure scenario for our experiments where we simulate node failure by shutting down the node from OpenStack as well as Linux's reboot command on the VM that is hosting the pod which is streaming the video. The measurements of this set of experiments for the architectures of Figure 4-7, Figure 4-5, and Figure 4-6 are shown in Table 4-4.

Table 4-4. Kubernetes with the Most Responsive Configuration – Service Outage due to Node Failure Scenario.

architecture	failure trigger (unit: seconds)	reaction time	repair time	recovery time	outage time
StatefulSet controller (Figure 4-7)	Node Shutdown	NA	NA	NA	NA
	Node Reboot	1.738	124.078	126.155	127.893
State Controller integrated with StatefulSet controller (Figure 4-5)	Node Shutdown	2.209	NA	0.727	2.936
	Node Reboot	1.970	128.201	0.883	2.853
State Controller integrated with Deployment controller (Figure 4-6)	Node Shutdown	2.033	3.825	0.780	2.813
	Node Reboot	2.050	4.513	0.918	2.968

4.1.3.4 *Evaluating the Availability Achievable with the State Controller for Stateful Microservice Based Applications Compared to Non-Kubernetes Based Solutions (RQ8)*

We address RQ8 to better position the availability results obtained with our solution. As mentioned in Sub-section 3.2.2.4, we consider the OpenSAF middleware as a proven solution for providing availability. In a previous work [44], a set of availability experiments with OpenSAF were conducted. These experiments covered different failure scenarios with the same video streaming application (VLC). For comparison, we consider the failure scenario of VLC component failure and physical host failure, corresponding to VLC container failure and node reboot in Kubernetes, respectively. In the experiments with OpenSAF, the application is stateful and the redundancy model is 2N [13]. The architecture for these experiments are shown in Figure 4-8. The application has two VLC components, one active and the other one as an instantiated standby to take over in case of failure of the active. The configuration with regards to node failure detection used is the default configuration of OpenSAF. The results of the experiments with OpenSAF are shown in Table 4-5.

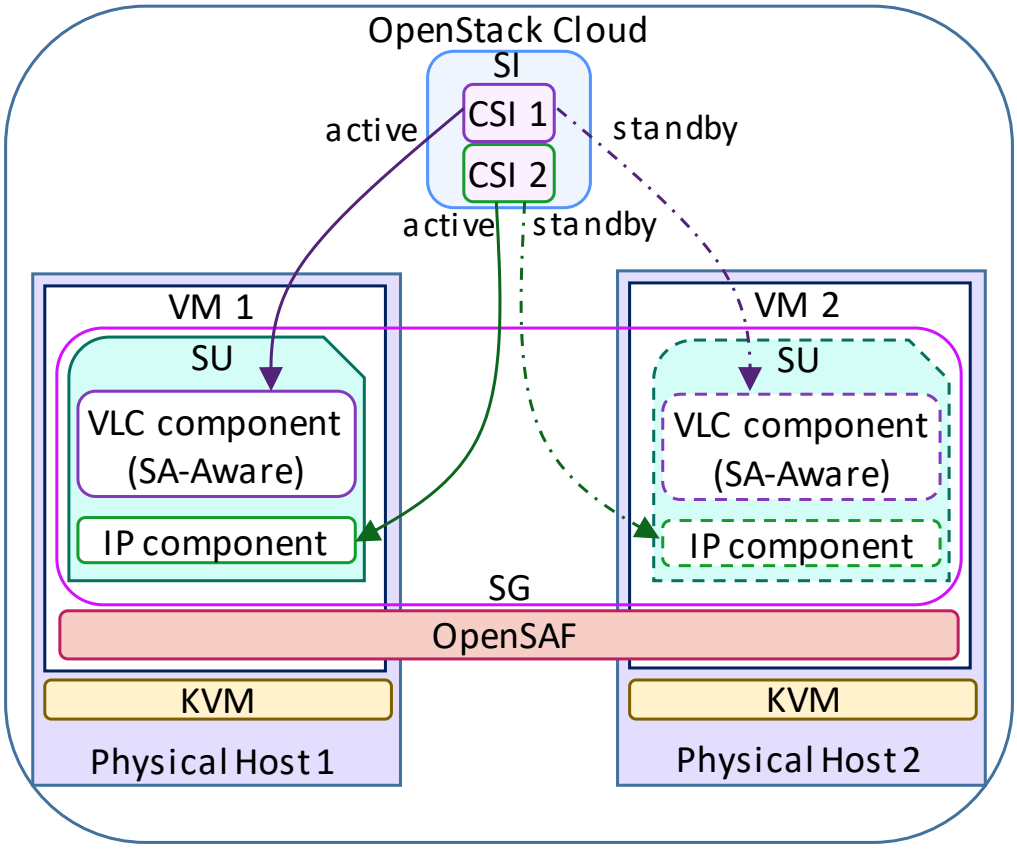

Figure 4-8. The architecture for availability experiments with OpenSAF (stateful VLC).

Table 4-5. Experiments with OpenSAF (SA-Aware VLC).

failure trigger (unit: seconds)	reaction time	repair time	recovery time	outage time
VLC Component Failure	0.089	0.181	0.140	0.229
VM Failure	3.233	21.074	0.116	3.344
Physical Host Failure	3.229	37.795	0.101	3.329

4.1.4 Analysis and Discussion

In this sub-section, we analyze the results of the experiments of Sub-section 4.1.3 in order to answer the research questions we brought earlier. We analyze the results for each failure scenario separately.

Analysis of Service Outage due to Application Container Failure Scenario: In this failure scenario, before killing the application container, the IP address of the pod was in the endpoints list and the service was available. After the failure, the service becomes unavailable. However, since Kubernetes has not detected the failure yet, the IP address of the pod stays in the endpoints list. The reaction time is when Kubernetes detects the failure and removes the pod's IP from the endpoints list. As it is observed in Table 4-1, Table 4-2, and Table 4-3, the reaction times of all architectures are measured between 0.549 and 0.739 seconds. The increase of reaction time from 0.679 seconds to 0.739 seconds shown in Table 4-1 and Table 4-2 can be considered as the overhead of integrating the State Controller with StatefulSet controllers on the reaction time. Also, the average repair time of the failed pod was measured between 1.027 to 1.052 seconds for all architectures. However, it is only the architecture in Figure 4-7 (StatefulSet controller without the State Controller) whose service recovery depends on the repair time and therefore, has the longest service recovery which is 1.480 seconds making the average service outage time 2.159 seconds with the standard deviation of 0.24 seconds. In the other architectures where the proposed State Controller is integrated with the StatefulSet controller and Deployment controller (Figure 4-5 and Figure 4-6), the service recovery does not depend on the repair time. The reason is that after the failure is detected by Kubernetes, that is, when it marks the pod's state "not ready", the State Controller fails over the service to the standby pod which already has the last stored state by only changing the HAState labels and there is no need to wait for the failed pod to be restarted. Therefore, the average service outage time for

the architecture of Figure 4-5 is less and is 1.400 seconds with the standard deviation of 0.42 seconds. The average service outage time for the architecture of Figure 4-6 is 1.205 seconds with the standard deviation of 0.30 seconds. In average, we observe a 55% improvement in recovery time when the proposed State Controller is integrated with Kubernetes.

To better position the impact of integrating the State Controller with Kubernetes on availability, we compare our results with those of the experiments conducted with OpenSAF. As shown in Figure 4-9, the OpenSAF solution shows a lower outage – only 0.229 seconds. Table 4-5 shows that the difference is in both reaction time and recovery time. The reason is

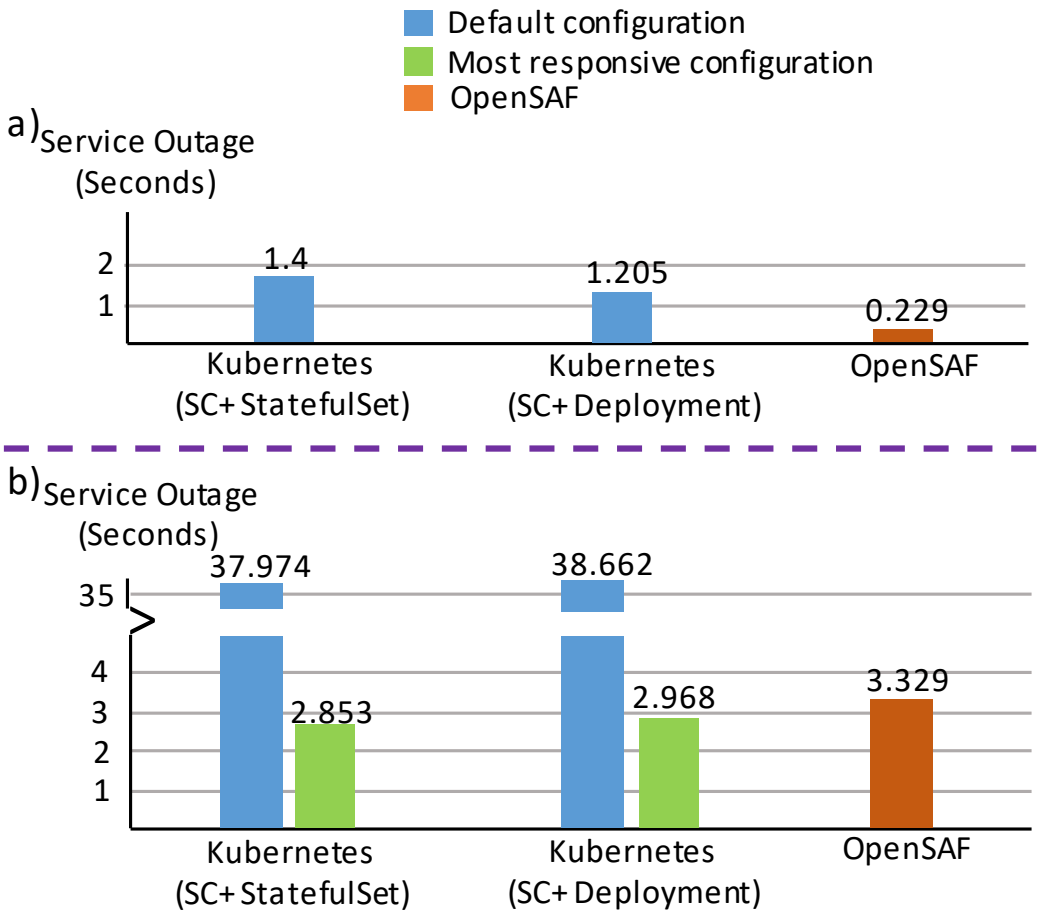

Figure 4-9. Comparing Kubernetes and OpenSAF from availability perspective. a) Application container/component failure scenario, b) Node/physical host reboot scenario.

that in the experiments with OpenSAF, the VLC component is modified and implements the AMF API and therefore its failure can be detected faster by the AMF.

Analysis of Service Outage due to Pod Process Failure Scenario: The pod process is an environment for managing a group of application containers. Although the containers of a pod communicate and share volumes through this environment, in the newer versions of Kubernetes, the failure of the pod process itself does not result in their failure. After the pod process failure, Kubernetes gives the application container 30 seconds to terminate. During this grace period, the service is available for incoming requests as well as for ongoing ones. After the 30 seconds, the application container is terminated and restarted along with the pod process. For the architecture of Figure 4-7, the failure event is when the application container is terminated which is the same as the reaction time. Therefore, it is zero seconds. Later, the Kubelet restarts the pod process and the application container and the pod is considered repaired when the restart procedure is finished which takes 0.943 seconds. Subsequently, the service is resumed 2.133 seconds after the failure making the average service outage 2.133 seconds with the standard deviation of 0.21 seconds. For the architectures of Figure 4-5 and Figure 4-6, the measurements correspond to different events. For these architectures, the failure event is induced by the State Controller and happens before the application container is terminated by the Kubelet. That is, when the State Controller detects that the Kubelet has reported a pod process failure and it assigns the standby HA state to the failed active pod and terminates all its active processes. The reaction for these architectures is the same as the failure time making the reaction time zero seconds. Recovery time is when the healthy pod has been assigned the active HA state and the video is resumed, which was measured between 0.656 seconds with the standard deviation of 0.14 seconds and 0.637 seconds with the standard deviation of 0.15 seconds for these architectures. However, the repair time depends on the graceful termination period

of the pod. The results show that integrating the State Controller reduced the outage time by around 70%.

Analysis of Service Outage due to **Node Failure Scenario:** In this failure scenario, as it is observed in Table 4-1, Table 4-2, and Table 4-3, with the default configuration of Kubernetes, it takes between 30 to 40 seconds for Kubernetes to consider the node as lost and remove the IP of its pods from the endpoints list. In the architectures where the application is deployed by a StatefulSet controller (Figure 4-7 and Figure 4-5), if the failed node does not become responsive and does not rejoin the cluster, Kubernetes will not recreate the pods of the failed node on other nodes; therefore there will be no repair time. This means that the service will not be recovered in the node shutdown scenario for the architecture of Figure 4-7. For the architecture in Figure 4-5, however, the State Controller will failover the service to the standby pod which is able to resume the service in 0.710 seconds on average with the average service outage of 37.946 seconds with the standard deviation of 2.72 seconds. For the architecture of Figure 4-6 with the same scenario, the measured recovery times is 0.760 seconds making the average service outage time 38.662 seconds with the standard deviation of 3.41 seconds which is close to that of the architecture in Figure 4-5. However, with this architecture, after the default pod eviction timeout, a new pod is recreated on another node making the repair time 262.932 seconds on average. We observe that in the scenario where service recovery was not possible when only using StatefulSet controllers, integrating the State Controller has enabled service recovery measured as 0.710 seconds.

When service outage is due to node reboot, that is, the node becomes responsive again as it rejoins the cluster, Kubernetes will be able to recreate the pod in all architectures. This means that repair time depends on how fast the node can become responsive again. As it is observed in Table 4-1, for the architecture in Figure 4-7, the repair time affects the recovery

time which was measured as 127.380 seconds resulting in the average service outage of 164.507 seconds and standard deviation of 6.86 seconds. However, for the architectures of Figure 4-5 and Figure 4-6, since the State Controller reassigns the active HAState to the standby pod and since service recovery does not need the failed pod to be repaired, the recovery time will not depend on the node reboot. Therefore, the average service outage time is reduced and is 38.460 seconds with a standard deviation of 4.460 seconds for the architecture of Figure 4-5. The average service outage time for the architecture of Figure 4-6 is 36.955 seconds with the standard deviation of 3.11 seconds. In the node reboot scenario, we observe a 99% improvement in the recovery time when the State Controller is integrated with Kubernetes.

We also reconfigured Kubernetes to its most responsive configuration and repeated the node reboot scenario for all three architectures. The measurements of Table 4-4 show that the new configuration has decreased service outage by 22% for the architecture of Figure 4-7 and by 92% for the architectures of Figure 4-5 and Figure 4-6. For the architecture of Figure 4-7, the new configuration only affects the reaction time. The repair time remains the same and depends on how fast the node is able to reboot. For the architecture in Figure 4-6, the new configuration changes both reaction and repair time. However, since in this architecture recovery does depend on the repair time, only the change in the reaction time affects the outage.

As shown in Figure 4-9, the service outage of the physical host failure scenario of the OpenSAF solution is significantly lower compared to that of the node reboot scenario with our proposed State Controller under the default configuration. The main contributing factor to the higher outage with Kubernetes is the node failure handling configuration parameters. Therefore with the most responsive configuration, the results of our solution integrated with Kubernetes are comparable with those of the OpenSAF solution.

In Section 2.2, it was mentioned that authors of [27] have considered a database as a microservice and proposed that the state replication between microservice instances can be done by the database process. However, one cannot guarantee service recovery and continuity only by replicating the state data between microservice instances. For service recovery, a microservice instance needs to be notified and ordered to access the replicated data and continue the service. Also, it should be clear where the state data of each microservice instance is stored. In our solution, a state replication mechanism is provided through which the active pod replicates its state data to the standby pod and a third party (the State Controller) notifies the standby pod if its corresponding active pod fails. Moreover, each pod stores its data separately and is aware of the location of its data.

4.1.5 Limitations of the Proposed State Controller

In the previous sub-section, we proposed a State Controller which improves the availability of stateful applications deployed with Kubernetes by integrating the concept of high-availability states, i.e., active and standby. However, there are some limitations to this solution in terms of handling elasticity. In the proposed solution, the State Controller only assigns one active HA state and one standby HA state. Meaning that if due to the increase in load, the StatefulSet (or Deployment) controller scaled out the application, the newly added pods will not be assigned any HA states. Even if the State Controller was able to assign HA states to the new pods, replicating the state data was not possible. Because the state replication service that is created before running the State Controller would have multiple standby pods as its endpoints and would send the state data of an active pod to several standby pods which makes it impossible for a standby pod to have all the state data of its active pod. The next section explains how the State Controller is modified to handle elasticity.

4.2 Handling Elasticity with the State Controller

In this section, we address the issue of the proposed State Controller in handling availability where the application is scaled in or out.

To address the problem mentioned in Sub-section 4.1.3, we modified our proposed State Controller so it can handle multiple HA state assignments when the application is scaled in or out. In this solution, the application can be deployed by a StatefulSet controller as well as a Deployment controller. Moreover, a service called the application service should be created that uses the HAState label and targets all pods with the active label. However, unlike the initial State Controller introduced in Section 4.1, we have more than one state replication service which are automatically created by the State Controller. In this solution, the State Controller holds pairs of active and standby pods and identifies a pair by adding a "peer" label to each pod which gets its corresponding active or standby pod's name. Below are the steps that the modified State Controller (i.e., the State Controller enriched with elasticity) takes to manage the availability of stateful applications (Figure 4-10). Note that the entrypoint process of pods introduced in Section 4.1 remains the same.

A. Sorts running pods based on their creation time
B. Picks first two pods
C. Assigns HA state and peer labels to pods (Figure 4-11)
    1) The first pod named "X" is assigned the active HA state and becomes an endpoint to the application service
    2) The second pod named "Y" is assigned the standby HA state
    3) Assigns peer label to pod "X" with the value of "Y" and to pod "Y" with the value of "X"

4) For the active pod named “X”, creates a service named “replicate-X” that selects a pod with HAState label equal to standby and peer label equal to “X”. Pod “X” periodically replicates its state data to a service named “replicate-X”

D. If there are more pods remaining, picks the next two pods and goes to step 3

E. Monitors the events of the API server

1) If the event corresponds to the service state of a pod changing into “not ready”

i. If the failed pod had the active HA state

- ✓ It assigns active HA state to the standby pod which was the peer of the failed active pod. The new active pod becomes the endpoint of the application service and restores the last state from its storage area in the PV and resumes the service.
- ✓ It assigns standby HAState to the failed pod and deletes the state replication service of the failed active pod
- ✓ Creates the replication service for the new active pod

ii. If the failed pod had the standby HAState, it ensures that the failed pod is assigned the standby HAState after it is repaired

2) If the event corresponds to a scale-out, then goes to step 3

3) If the event corresponds to a scale-in, it deletes the state replication service for a deleted active-standby pair

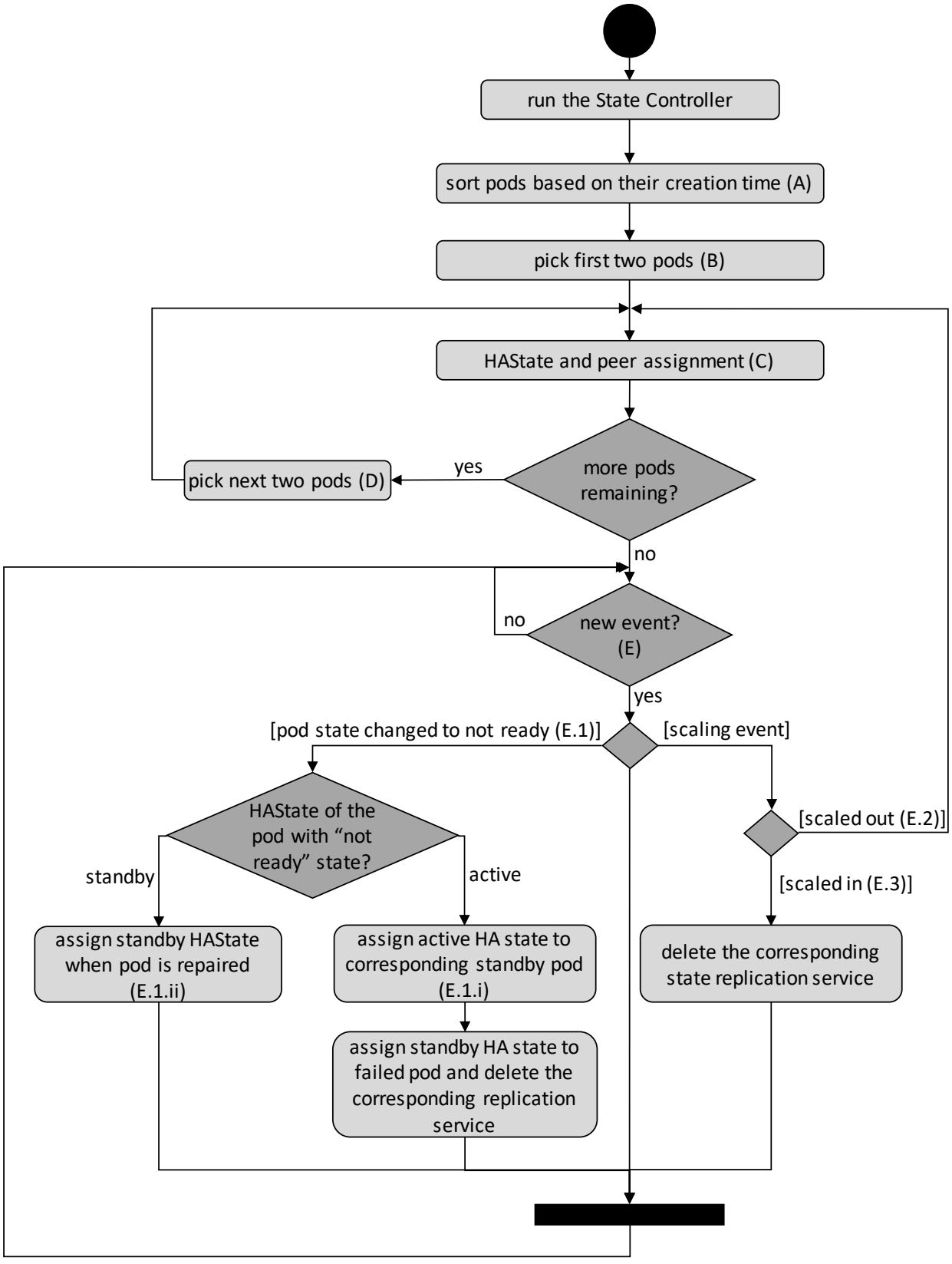

Figure 4-10. The behavior of the State Controller enriched with elasticity.

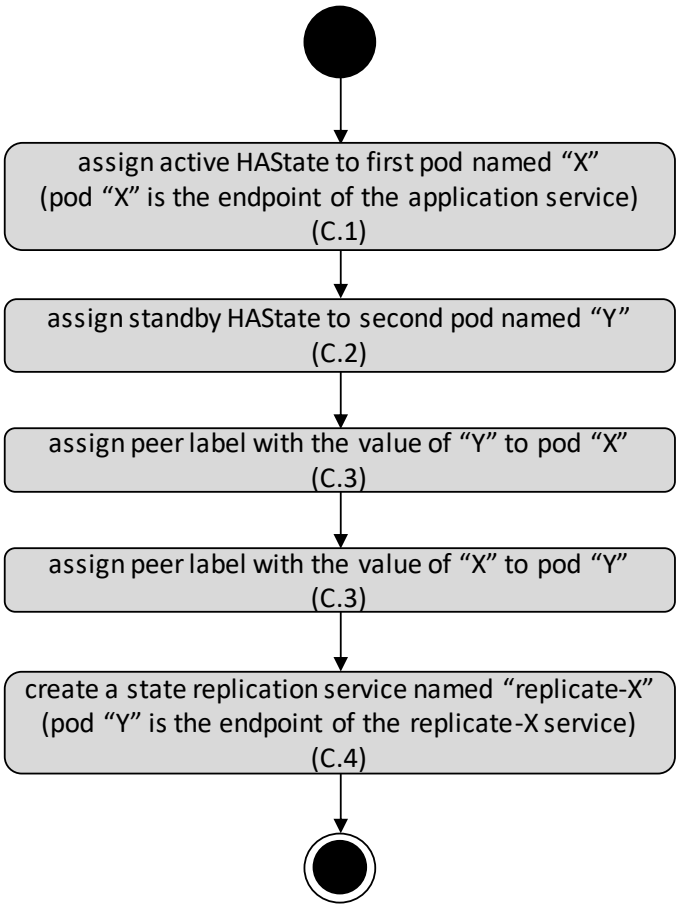

Figure 4-11. Setting the HAState and Peer label and variables to pods (Step C).

The modified State Controller has been implemented as a proof of concept and developed using the Go programming language [49] and the client-go library [50] of Kubernetes. It is important to mention that we assume that an external elasticity engine is present and makes the scaling decisions. We also assume that in the scale-out events, the number of pods is increased by two, and in scale-in events, it is decreased by two as well.

Figure 4-12 shows an example architecture where we integrate the modified State Controller with Kubernetes. In this architecture, the application is video-on-demand (VoD) streaming and is deployed by a StatefulSet controller. As it is shown in this architecture, a service called the application service should be created which only selects the pods with HAState label equal to active and app label equal to VoD and the State controller should be running.

In the architecture of Figure 4-12, when the StatefulSet controller is deployed, it creates two pod replicas named VoD-0 and VoD-1. In the beginning, there are no pods in the endpoints list of the application service. After the pods are deployed, the State Controller assigns the active HA state to VoD-0 which makes it an endpoint to the application service. Once VoD-0 receives a request from a client for streaming a video, it periodically checkpoints its streaming position to its PV. In the checkpointing process, in addition to storing the state in PV0, VoD-0 also replicates its state data to a service called "replicate-VoD-0". This service is created by the State Controller after it assigns the standby HA state to VoD-1 as well as the peer label with the value of "VoD-0". Note that the "replicate-VoD-0" service selects a pod with HAState label equal to standby, app label equal to VoD, and peer label equal to VoD-0. This way, VoD-0 and VoD-1 are paired, and VoD-1 stores the state data of its active pod (VoD-0) in its PV.

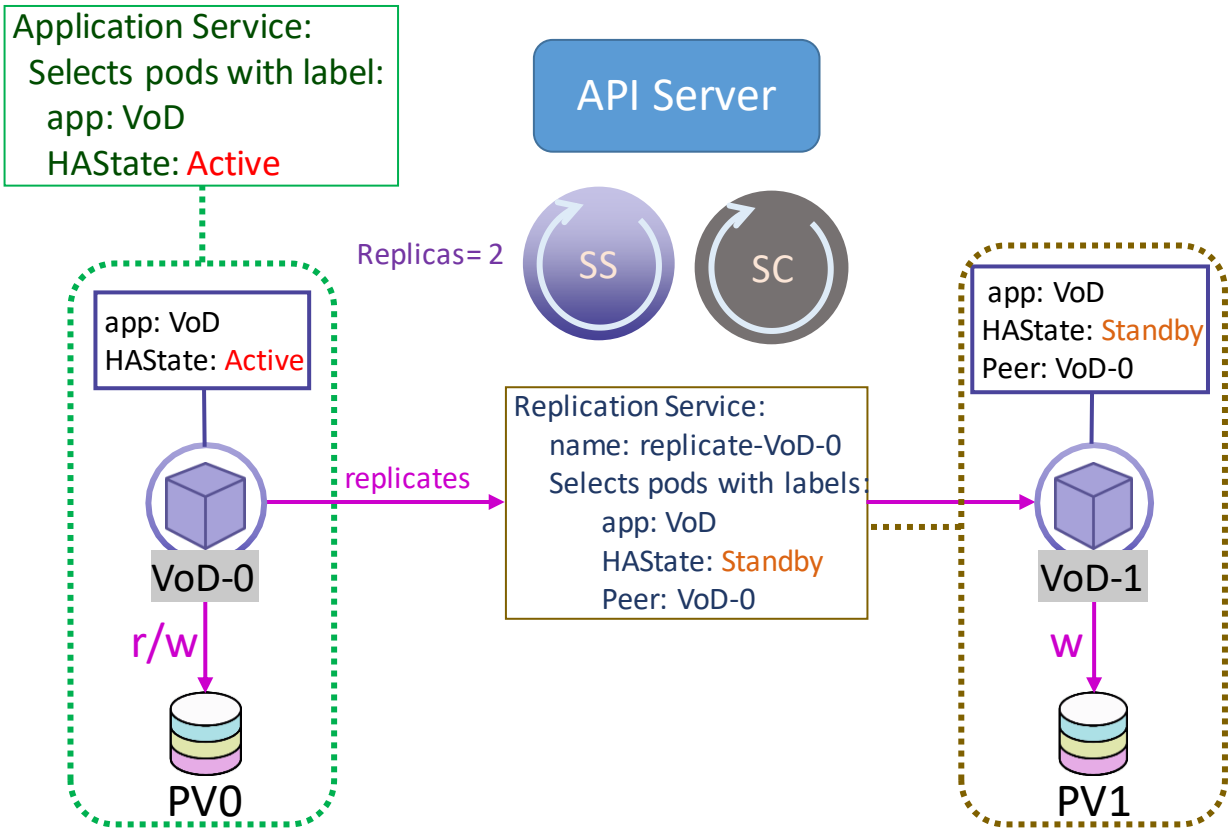

Figure 4-12. An example architecture for the State Controller enriched with elasticity.

Let us assume that the demand for service increases and the StatefulSet is ordered to add n-2 more pods (Figure 4-13). The State Controller detects that pods VoD-2, VoD-3... VoD-(n-1) are added. It assigns the active HA state to VoD-2 making it the second endpoint to the application service. Moreover, it assigns the standby HA state to VoD-3 and adds a peer label with the value of VoD-2. This will be done until all pods are assigned an HA state. In the next step, for each pair of active-standby, the State Controller creates a service named "replicate-{active pod's name}" which selects the standby pod as its endpoint. Through the "replicate-{active pod's name}" service, the active pod replicates its state data to its corresponding standby pod. Note that the modified State Controller can integrate with Deployment controllers as well and maintain multiple pairs of active-standbys (Figure 4-14).

Now let us assume that the active pod VoD-(n-2) fails due to application container failure. The first step for the State Controller after detecting that there was a failure is to find the peer of VoD-(n-2). The State Controller holds the pairs of pods in an array and finds that the corresponding standby pod for VoD-(n-2) is VoD-(n-1). Therefore, it fails over the service to VoD-(n-1) by changing its HAState label and variable into active. It also changes the HAState label and variable of VoD-(n-2) into standby after it is repaired. Moreover, it deletes the "replicate-VoD-(n-2)" service and instead, creates a service named "replicate-VoD-(n-1)" through which the new active pod (VoD-(n-1)) can replicate its state data to the new standby pod (VoD-(n-2)) after it is repaired.

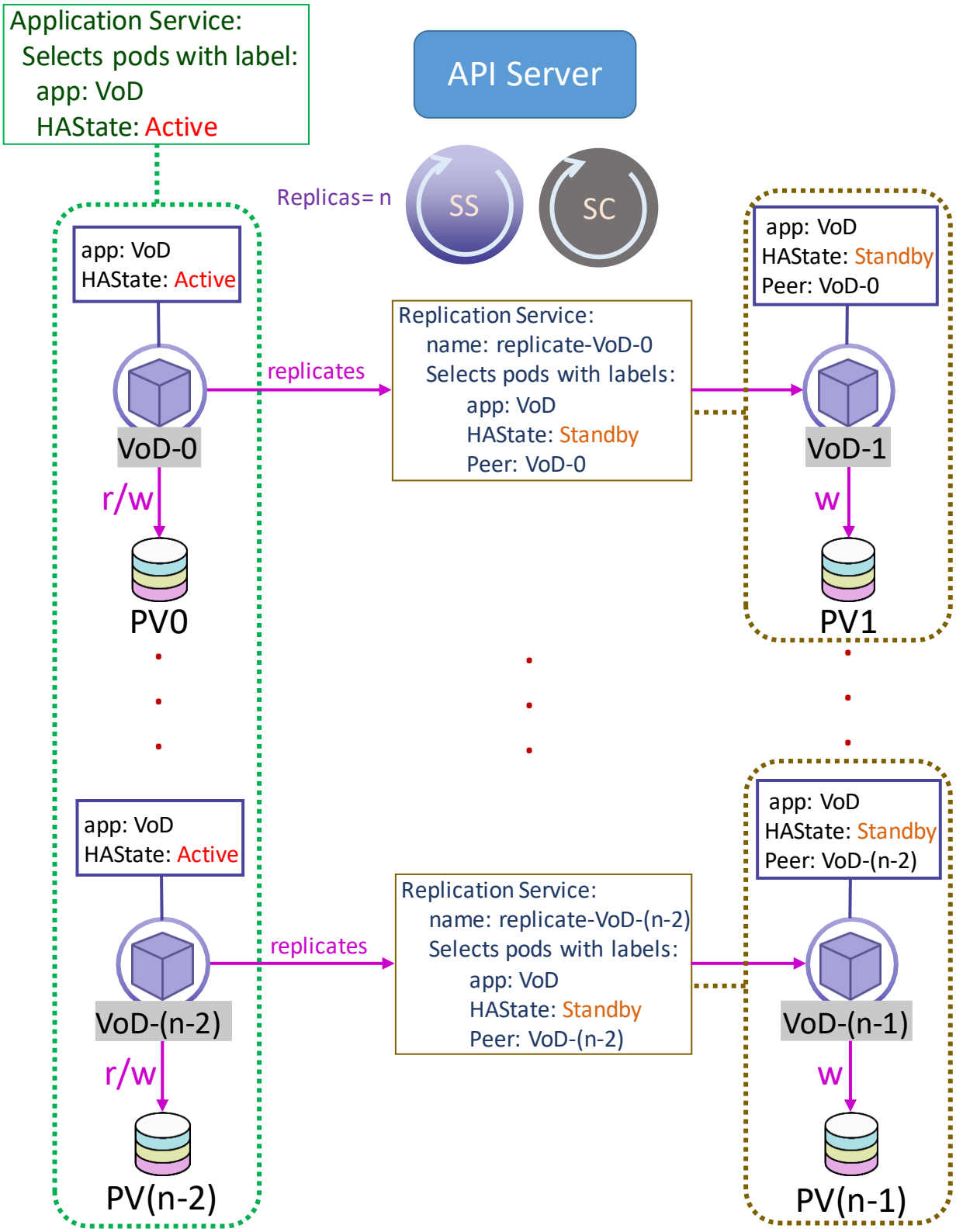

Figure 4-13. Modified State Controller integrated with StatefulSet controller - stateful application with multiple pairs of active-standbys.

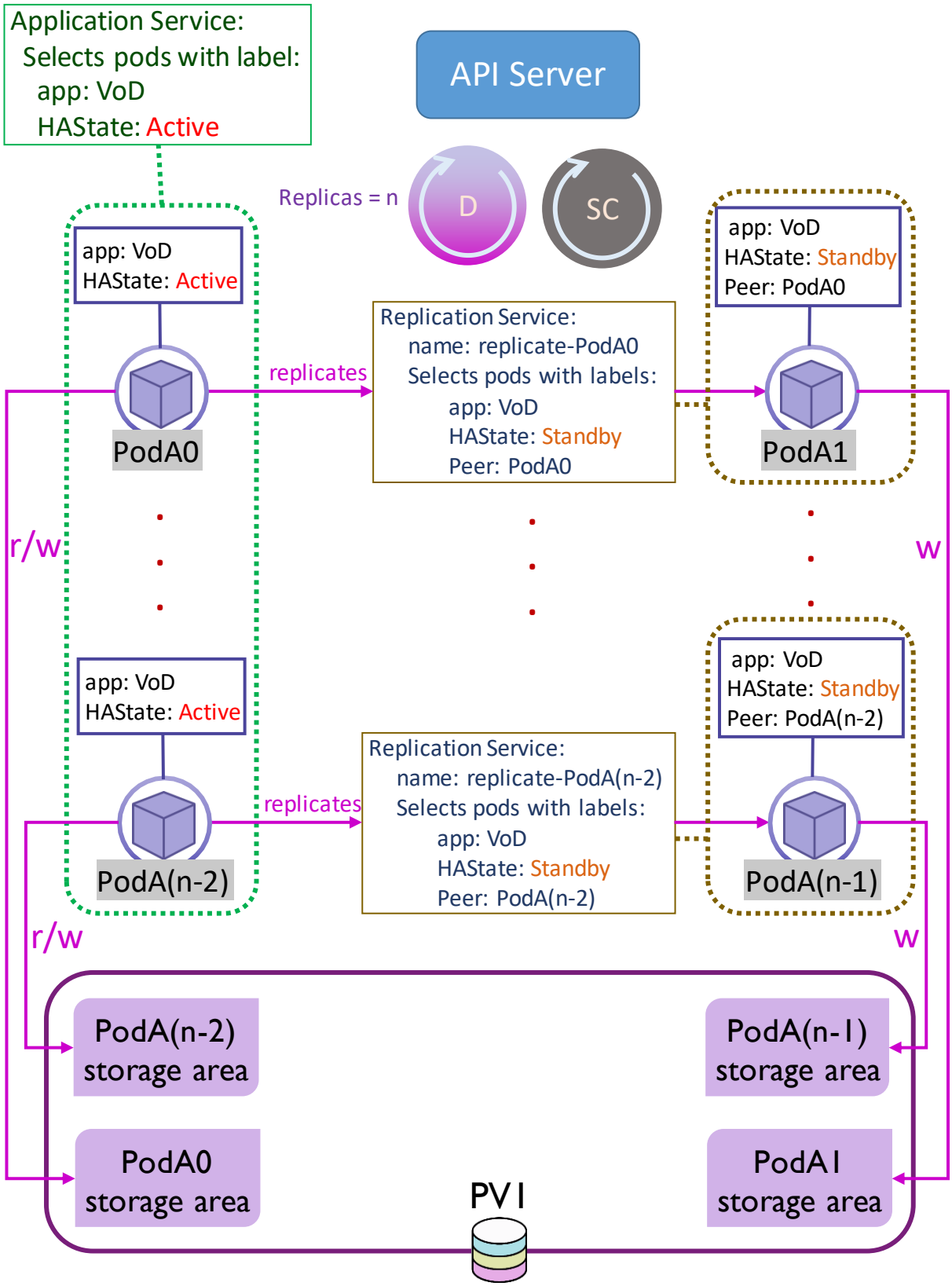

Figure 4-14. Modified State Controller integrated with Deployment controller - stateful application with multiple pairs of active-standbys.

4.2.1 Evaluating the Scaling Overhead and the Achievable Service Availability with the Modified State Controller

In the Sub-section 4.1.3, we evaluated the achievable service availability when the initial State Controller (the State Controller than can only keep one pair of active-standby) is integrated with Kubernetes. In this sub-section, we evaluate the achievable availability as well as the scaling overhead of integrating the modified State Controller (i.e., the State Controller enriched with elasticity) with Kubernetes. We do these evaluations by addressing the following research questions:

RQ9: What is the impact of integrating the modified State Controller on the provided availability?

RQ10: What is the impact of scaling during failover on the availability that the modified State Controller can provide for its managed microservices?

RQ11: What is the scaling overhead of integrating the modified State Controller?

RQ12: What is the impact of simultaneous failure of multiple active pods on the outage time for each failed pod?

To address these research questions, we conducted a set of experiments and measured the following metrics:

1. Availability metrics defined in Sub-section 3.2.1.1.
2. Scaling time: The time between when the scaling request is sent until the last pod is deployed and ready (or deleted in case of scale-in).
3. HA state assignment time: The time between when the scaling request is sent until the State Controller assigns HA state to the last added pod.

In the experiments where we measure the availability metrics, the failure scenario is service outage due to application container failure. The scaling decision is not made by the State Controller and we assume that the application is scaled in or out by two. Moreover, the experiments' setting is the same as discussed in Sub-section 4.1.3. However, instead of two worker nodes, we have eight worker nodes in these experiments in order to be able to scale-out the application. The State Controller used in these experiments is the modified version which can handle multiple active and standby HA state assignments.

4.2.1.1 *Evaluating the Impact of Integrating the Modified State Controller on the Provided Availability (RQ9)*

In Sub-section 3.1.2, we presented one architecture with StatefulSet controller (Figure 3-2) and another with Deployment controller (Figure 3-3) for deploying stateful microservice based applications with Kubernetes. We discussed that with both architectures, other pods cannot recover the service for a pod when it fails. Because the healthy pods do not know about the failure, nor are aware of the location its data are stored. It was also mentioned that with the architecture of Figure 3-3, we cannot always rely on the restart procedure for recovering the stored service state. Because, the identity of a restarted pod changes in certain failure scenarios such as node failure scenarios. However, in the experiments in this sub-section, we only consider the failure scenario of service outage due to application container failure. Therefore, we consider both architectures of Figure 3-2 and Figure 3-3 to measure the availability metrics as a baseline and compare with the availability that integrating the modified State Controller can provide. Figure 4-15 shows the concrete architectures for the experiments where we evaluate Kubernetes in terms of the availability it provides for stateful applications only by its repair actions through a set of availability experiments. The measurements for these experiments are shown in Table 4-6.

To answer this research question, similar to the experiments in Sub-section 4.1.3.2, we evaluate the impact of integrating the State Controller on the availability by measuring the availability metrics through a set of availability experiments. The architectures for these experiments are depicted in Figure 4-5 and Figure 4-6 and the failure scenario is service outage due to application container failure. However, the State Controller used in these experiments is the modified version which can hold pairs of active-standbys. The measurements of this set of experiments are shown in Table 4-7.

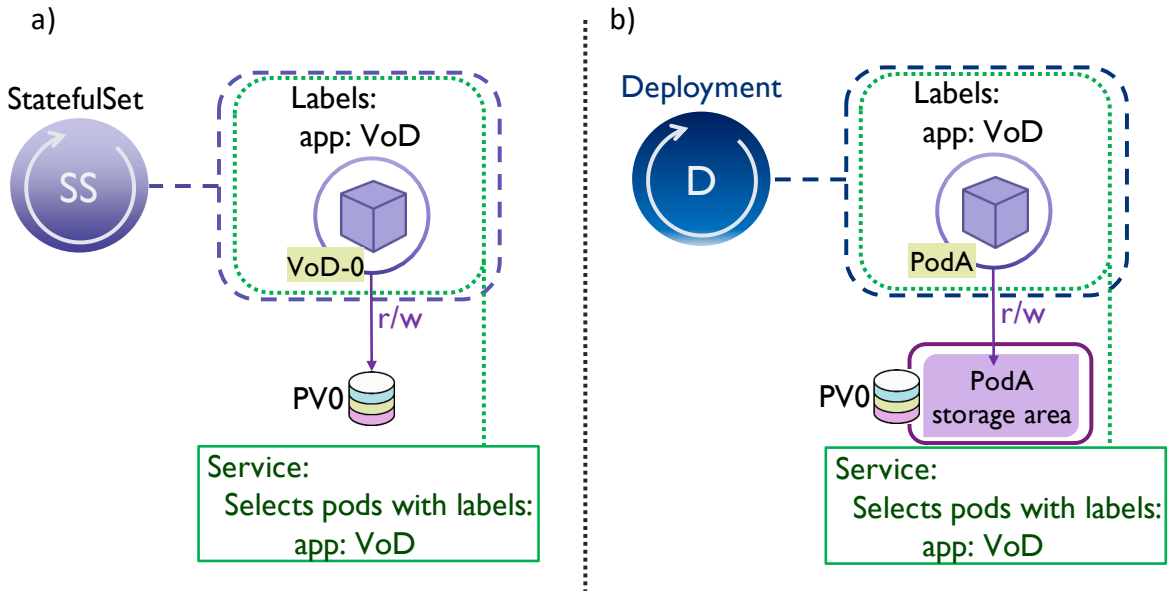

Figure 4-15. Concrete architecture for experimenting with Kubernetes - Stateful microservice based application with No-Redundancy redundancy model. a) Deployed with StatefulSet controller. b) Deployed with Deployment controller.

Table 4-6. Evaluating the repair actions of Kubernetes for providing availability – Application container failure scenario.

architecture (unit: seconds)	reaction time	repair time	recovery time	outage time
StatefulSet controller (Figure 4-15 (a))	0.679	1.029	1.480	2.159
Deployment controller (Figure 4-15 (b))	0.554	1.021	1.534	2.088

Table 4-7. Evaluating the modified State Controller for providing availability – Application container failure scenario.

architecture n = 2 (unit: seconds)	reaction time	repair time	recovery time	outage time
Modified State Controller integrated with StatefulSet controller (Figure 4-13)	0.719	1.083	0.793	1.512
Modified State Controller integrated with Deployment controller (Figure 4-14)	0.784	1.244	0.688	1.472

4.2.1.2 *Evaluating the Provided Availability by the Modified State Controller when Failover and Scaling Overlap (RQ10)*

In this research question (RQ10), we are interested in evaluating the impact of a simultaneous scaling event on the availability that the State Controller can provide for its managed application. To address RQ10, we conduct a set of experiments under two scenarios; scale-out and scale-in. For the scale-out scenario, we consider both architectures of Figure 4-5 and Figure 4-6 where the modified State Controller is integrated with a StatefulSet controller and a Deployment controller, respectively. In these architectures, two pods are deployed (one active and one standby) and we forcefully kill the application container of the active pod that is streaming a video. While the service is being recovered by the State Controller, we scale the application to four pods. We measure the availability metrics for the failed pod as well as the scaling time and HA state assignment time for the added pods. We compare the availability metrics of this set of experiments with those of an experiment where no scaling event had happened during failover. Moreover, we compare the scaling time and HA state assignment time of this set of experiments with those of an experiment where no failure had happened during scaling the application. The measurements for the scale-out scenario are shown in Table 4-8.

In the scale-in scenario, four pods are deployed (two active-standby pairs) and we forcefully kill the application container of the active pod that is streaming a video and was created before the other active pod. Also, we have set the graceful termination period of pods to zero. Meaning that when a pod is ordered to be deleted, it will be done immediately. Note that in this scenario, we only consider the architecture of Figure 4-13 where the stateful application is deployed by a StatefulSet controller (n=4) and we do not consider the architecture where the application is deployed by a Deployment controller. Because, with Deployment controllers, there is no guaranteed order in deleting the pods in case of a scale-in. For example, in Figure

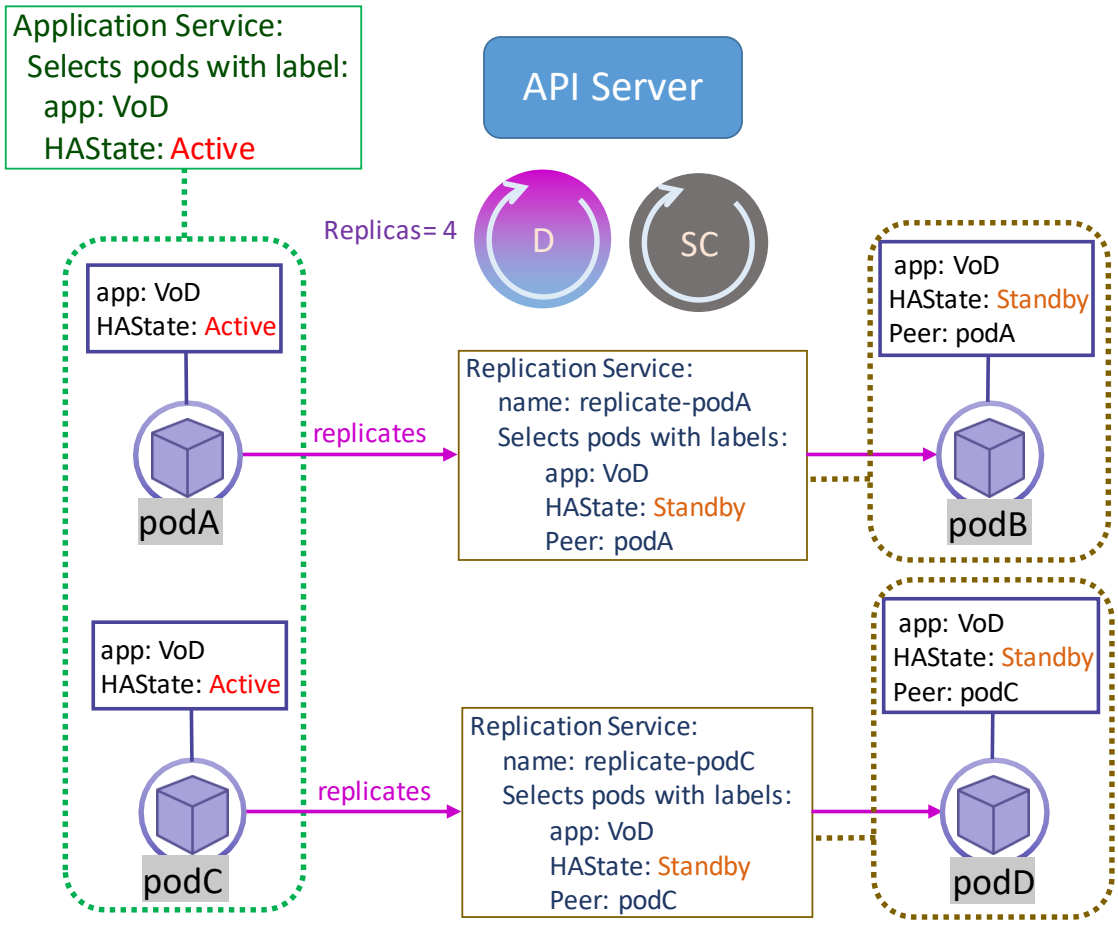

Figure 4-16. Example of integrating the modified State Controller with Deployment controllers.

4-16, let us assume that the active pod "podA" fails and during its recovery, the application is scaled in to two pods. Since the application is deployed by a Deployment controller, one possibility is that podB and podC are deleted. Therefore, there will be no standby pod for the failed active pod (podA) and service recovery cannot happen. Therefore we do not consider this architecture as it cannot guarantee service recovery.

In the scale-in scenario, we measure the availability metrics for the failed pod as well as the scaling time for the deleted pod. We compare the availability metrics of these experiments with those of an experiment where no scaling event had happened during failover. Moreover, we compare the scaling time of this set of experiments with those of an experiment where

no failure had happened during scaling the application. The measurements for the scale-out scenario are shown in Table 4-9.

Table 4-8. Evaluating the provided availability by the modified State Controller when failover and scaling overlap – Scale-out scenario.

architecture n = 2 (unit: seconds)	scenario	reaction time	repair time	recovery time	outage time	scaling time	HA state assign-ment time
Modified State Controller integrated with StatefulSet controller (Figure 4-13)	active pod fails	0.719	1.083	0.793	1.512	NA	NA
	application scaled out to 4	NA	NA	NA	NA	4.234	5.653
	scaling and failover overlap	0.689	1.161	1.012	1.701	7.049	7.293
Modified State Controller integrated with Deployment controller (Figure 4-14)	active pod fails	0.784	1.244	0.688	1.472	NA	NA
	application scaled out to 4	NA	NA	NA	NA	3.016	3.060
	scaling and failover overlap	0.607	1.205	1.028	1.635	5.055	5.608

Table 4-9. Evaluating the provided availability by the modified State Controller when failover and scaling overlap – Scale-in scenario.

architecture n = 4 (unit: seconds)	scenario	reaction time	repair time	recovery time	outage time	scaling time
Modified State Controller integrated with StatefulSet controller (Figure 4-13)	active pod fails	0.719	1.083	0.793	1.512	NA
	application scaled in to 2	NA	NA	NA	NA	0.712
	scaling and failover overlap	0.581	1.468	1.172	1.754	0.797

4.2.1.3 *Evaluating the Scaling Overhead of Integrating the Modified State Controller (RQ11)*

In this research question (RQ11), we are interested in evaluating the impact of integrating the State Controller on the time it takes for the application to be scaled. To address RQ11, we conduct a set of experiments under two scenarios; scale-out and scale-in. Also, we have set the graceful termination period of pods to zero. For the scale-out scenario, we consider the four architectures below.

- Figure 3-2, where the application is deployed by a StatefulSet controller.
- Figure 3-3, where the application is deployed by a Deployment controller.
- Figure 4-13, where the application is deployed by a StatefulSet controller with the Modified State Controller integrated.
- Figure 4-14, where the application is deployed by a Deployment controller with the Modified State Controller integrated.

In all architectures, the number of pods initially deployed is two (n = 2). In each round of the experiment, we scale the application from two pods to i pods where i gets one of the values in {4, 8, 16, 32, 64, and 128}. For this scenario, we measure the scaling time as well as HA state assignment time. The measurements for this set of experiments are shown in Table 4-10.

For the scale-in scenario, we consider the abovementioned architectures. However, in each round of the experiment, the number of pods initially deployed (i.e., n) gets one of the values in {4, 8, 16, 32, 64, and 128}. In each round of the experiment, we scale the application into 2 pods. Let us take the architecture of Figure 3-2 as an example. When n is equal to 128,

the StatefulSet, in the beginning, creates 128 pods and in the experiment we scale the application in into two pods. The measured metric for this scenario is the scaling time shown in Table 4-11.

Table 4-10. Scaling overhead and HA state assignment time for the scale-out scenario.

architecture (n = 2)	metric (unit: seconds)	2 to 4	2 to 8	2 to 16	2 to 32	2 to 64	2 to 128
StatefulSet controller (Figure 3-2)	scaling time	4.099	15.056	47.722	119.674	297.470	753.045
Modified State Controller integrated with StatefulSet controller (Figure 4-13)	scaling time	4.234	16.692	49.937	131.726	312.037	842.068
	HA state assignment time	5.653	17.018	51.865	133.373	316.107	845.114
Deployment controller (Figure 3-3)	scaling time	2.979	4.459	7.956	15.237	31.574	80.694
Modified State Controller integrated with Deployment controller (Figure 4-14)	scaling time	3.016	4.914	8.856	17.428	35.248	92.240
	HA state assignment time	3.060	6.763	16.142	35.290	73.001	147.798

Table 4-11. Scaling overhead for the scale-in scenario.

architecture n = {4, 8, 16, 32, 64, and 128}	metric (unit : seconds)	4 to 2	8 to 2	16 to 2	32 to 2	64 to 2	128 to 2
StatefulSet controller (Figure 3-2)	scaling time	0.555	1.353	2.613	5.459	11.440	26.062
Modified State Controller integrated with StatefulSet controller (Figure 4-13)		0.712	1.512	3.148	6.407	14.463	48.662
Deployment controller (Figure 3-3)		0.566	0.827	1.370	1.944	3.375	7.007
Modified State Controller integrated with Deployment controller (Figure 4-14)		0.641	1.327	1.555	2.375	4.441	8.821

4.2.1.4 *Evaluating the Impact of Simultaneous Failure of Multiple Active Pods on the Outage Time of each Failed Pod (RQ12)*

To address RQ12, we are interested in evaluating the State Controller in terms of availability when multiple active pods fail at the same time. Meaning that a second failure happens when the State Controller is still in the process of failover for the previously failed pod. In this research question (RQ12), we consider the architectures of Figure 4-13 and Figure 4-14 where the modified State Controller is integrated with a StatefulSet controller and a Deployment controller, respectively. For each architecture, the number of deployed pods is equal to 10. In these experiments, we forcefully kill the application container of *i* active pods where *i* can get the values in {1, 2, 3, 4, and 5}. In each round of the experiment, we measure the availability

Table 4-12. Availability metrics of simultaneously failed pods – The Modified State Controller integrated with a StatefulSet controller (Figure 4-13).

number of simultaneously failed active pods	Order number of the active pod whose failure was detected	reaction time	repair time	recovery time	outage time
1	first failed pod	1.180	1.437	1.312	2.491
2	first failed pod	0.646	2.205	2.208	2.855
	second failed pod	1.122	2.043	2.206	3.328
3	first failed pod	0.487	1.800	1.728	2.215
	second failed pod	0.791	1.799	2.067	2.858
	third failed pod	1.182	1.855	1.799	2.981
4	first failed pod	0.526	1.786	1.949	2.475
	second failed pod	0.855	1.708	1.935	2.790
	third failed pod	1.269	1.952	2.265	3.534
	fourth failed pod	2.720	2.382	2.101	4.820
5	first failed pod	0.473	1.605	2.220	2.693
	second failed pod	0.940	1.213	1.973	2.913
	third failed pod	1.240	2.189	2.250	3.491
	fourth failed pod	1.224	4.296	2.862	4.086
	fifth failed pod	1.800	4.574	4.010	5.810

metrics for each failed pod and compare how simultaneous failure of multiple active pods affects the service recovery. The results of this set of experiments are shown in Table 4-12 and Table 4-13.

Table 4-13. Availability metrics of simultaneously failed pods – The Modified State Controller integrated with a Deployment controller (Figure 4-14).

number of simultaneously failed active pods	Order number of the active pod whose failure was detected	reaction time	repair time	recovery time	outage time
1	first failed pod	0.749	0.454	1.886	2.634
2	first failed pod	0.642	0.464	1.863	2.505
	second failed pod	1.332	0.562	2.003	3.335
3	first failed pod	0.411	0.456	1.786	2.197
	second failed pod	1.006	0.538	1.924	2.930
	third failed pod	1.627	1.139	2.092	3.719
4	first failed pod	0.761	0.414	2.171	2.932
	second failed pod	0.889	0.702	2.084	2.974
	third failed pod	1.172	1.433	2.191	3.363
	fourth failed pod	2.446	2.172	2.587	5.034
5	first failed pod	0.495	0.558	2.096	2.591
	second failed pod	0.851	0.898	2.304	3.155
	third failed pod	1.163	1.562	3.117	4.280
	fourth failed pod	1.494	2.359	3.118	4.612
	fifth failed pod	2.471	2.936	3.394	5.865

4.2.2 Analysis and Discussion

In this sub-section, we analyze the results of the experiments of Sub-section 4.2.1 in order to answer the research questions we asked earlier.

In RQ9, we are interested in evaluating the impact of integrating the modified State Controller with Kubernetes on the provided availability. To address this research question, we conducted some availability experiments whose results (Table 4-6 and Table 4-7) show that integrating the State Controller improves service recovery is by around 50%. The reason is that with the State Controller, we no longer need to wait for the failed pod to be repaired in order to have the service recovered. The State Controller is able to recover the service faster by failing over to the Standby pod. However, we observe that integrating the State Controller has added an average overhead of 22% to the reaction time.

In the experiments for RQ10, we evaluate the impact of scaling during failover on the provided availability by comparing the measured availability metrics to those of the experiment where the only event is the failure (without any simultaneous scaling). The results of these experiments (Table 4-8 and Table 4-9) show that when a scaling event happens during recovery, the outage time is increased by 12% and 16% for the scale-out and scale-in scenario, respectively. We also evaluate the impact of scaling during failover on the scaling time by comparison to the experiments where the only event is the scaling (without any simultaneous failure). The results of the experiments for both scale-out and scale-in scenario (Table 4-8 and Table 4-9) show that when a failure happens during scaling, the scaling time is increased by 66% in the scale-out scenario and 12% in the scale-in scenario. Moreover, for the scale-out scenario, the HA state assignment time is increased by 56% on average. The reason is that when scaling and failover overlap, the State Controller is busy with failover and it can only

assign the HA states with some delay. We also discussed that with the architecture where the modified State Controller is integrated with a Deployment controller Figure 4-16, a scale-in request may result in deleting the standby pod of an active pod. Because with this architecture, there is no order in deleting the pods when the application scales in. Therefore, it is not possible to guarantee service recovery.

In RQ11, we are interested in evaluating the impact of integrating the modified State Controller with Kubernetes on the scaling overhead. To address this research question, we conducted experiments under the scale-in and scale-out scenarios. For the scale-out scenario (Table 4-12), when the application is deployed by a StatefulSet, integrating the modified State Controller has a scaling overhead of 7.5% on average. Also, integrating the modified State Controller with a Deployment controller increases the scaling time by 10.5%. The standard deviation for these measurements does not go above 23% of the average. Moreover, as it is observed in Figure 4-17 and Figure 4-18, the applications deployed by a Deployment controller have a shorter scaling time and HA state assignment time compared to the ones deployed by StatefulSets. The reason is that the pods deployed by Deployment controllers are created in parallel while with StatefulSet controllers, they should be created in an ordered manner which can take more time. While fast start-up time can be considered as a benefit of deploying the applications with Deployment controllers, one should take into consideration that service recovery is not guaranteed with Deployment controllers in scale-in scenarios. Because Deployment controllers do not scale-in the application in a guaranteed manner and the standby of an active pod might be deleted in the scale-in process while the active pod remains in the pods' list. We also conducted the experiments for the scale-in scenario whose results (Table 4-13) show that the scaling overhead of integrating the modified State Controller with StatefulSet controllers and Deployment controllers is 31% and 27% in average, respectively. The standard deviation for these

measurements does not go above 28% of the average. Similar to the scale-out scenario, we also observe in Figure 4-19 that the applications deployed with Deployment controller have a shorter scaling time. The reason is that the pods deployed with Deployment controllers are deleted in parallel while with StatefulSet controllers, they are deleted in an order which can take more time.

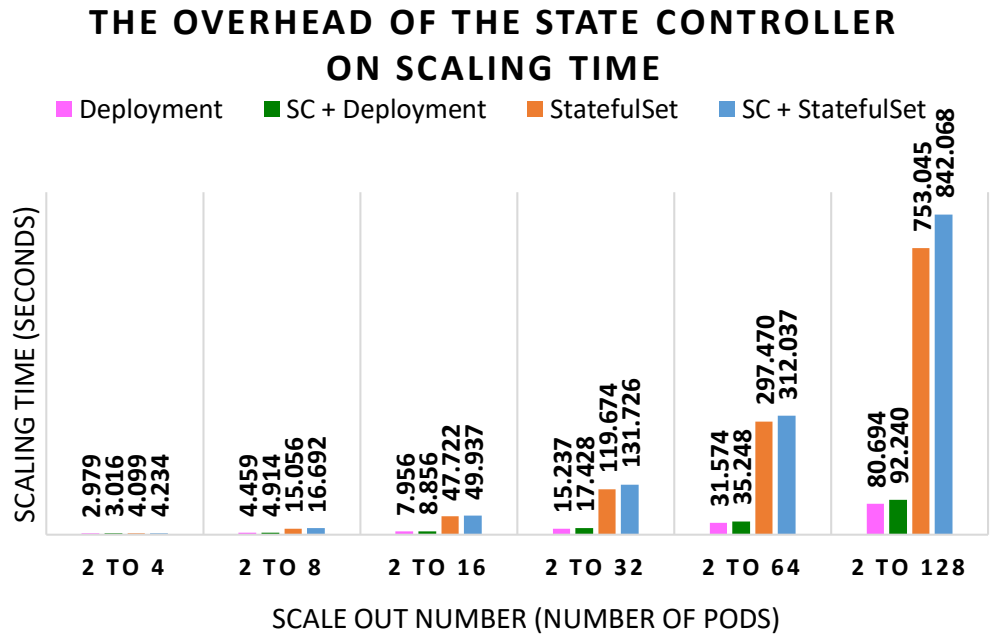

Figure 4-17. Scaling time results for experiments of RQ11 – Scale-out scenario.

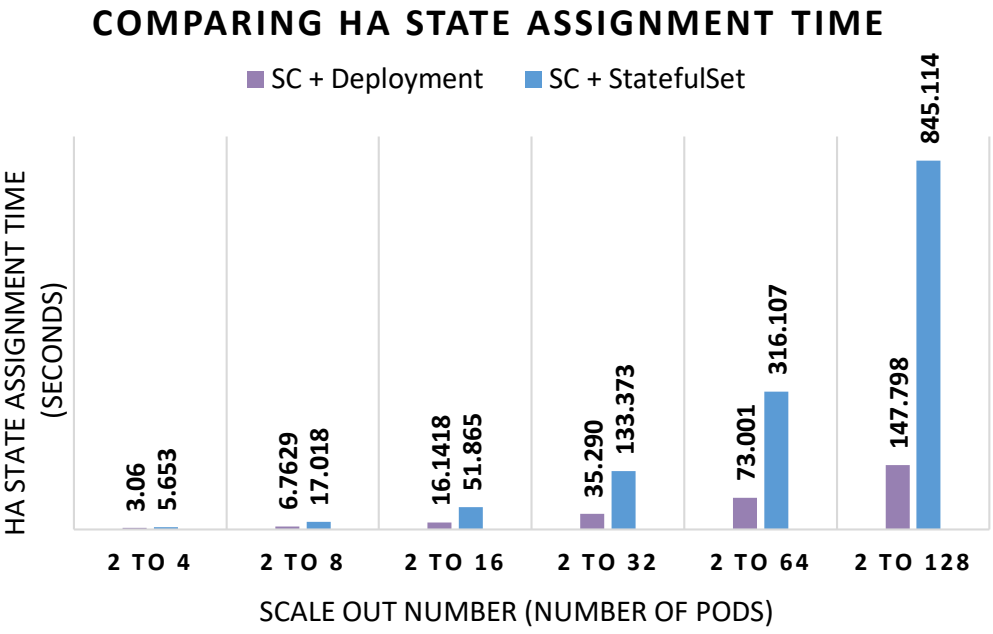

Figure 4-18. HA state assignment time results for experiments of RQ11 – Scale-out scenario.

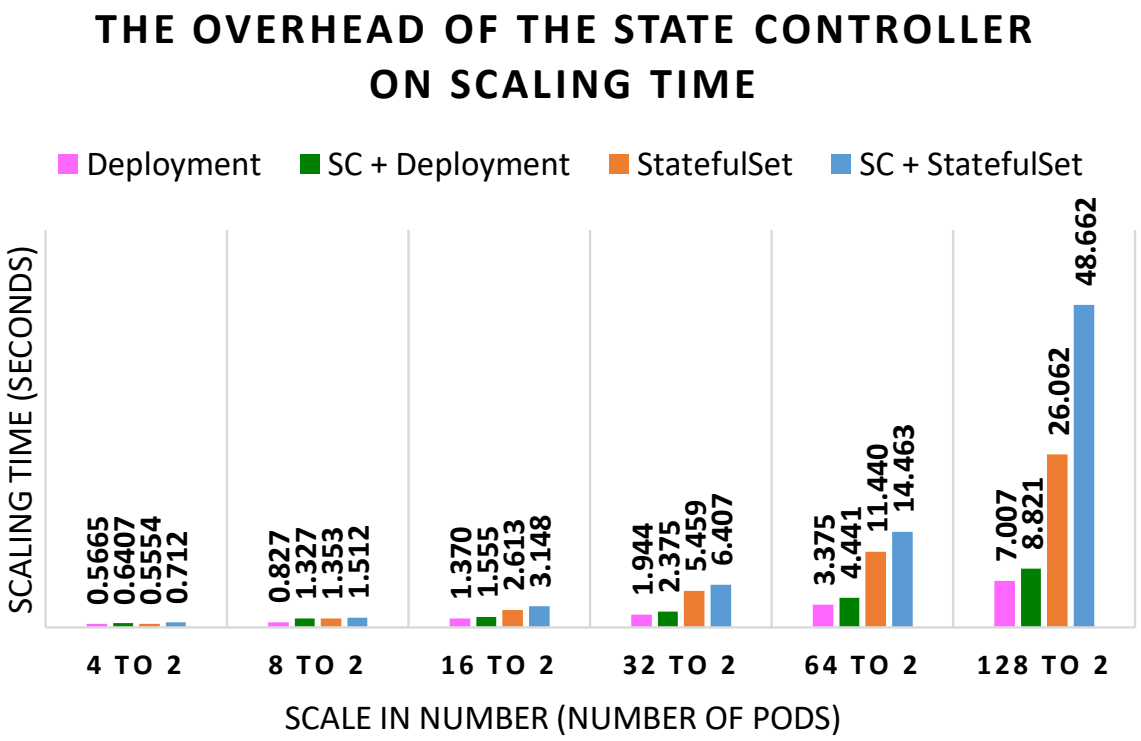

Figure 4-19. Scaling time results for experiments of RQ11 – Scale-in scenario.

In RQ12, we evaluate the availability provided by the modified State Controller when multiple pods fail simultaneously. The diagrams of Figure 4-20 and Figure 4-21 show that when multiple pods fail simultaneously, the later the pod's failure is detected, the longer it takes for the State Controller to recover the service for that pod. The reason is that once a pod's failure is detected, it is put as an event in a queue and its service will be recovered after the recovery of other pods' that were inserted in the queue before.

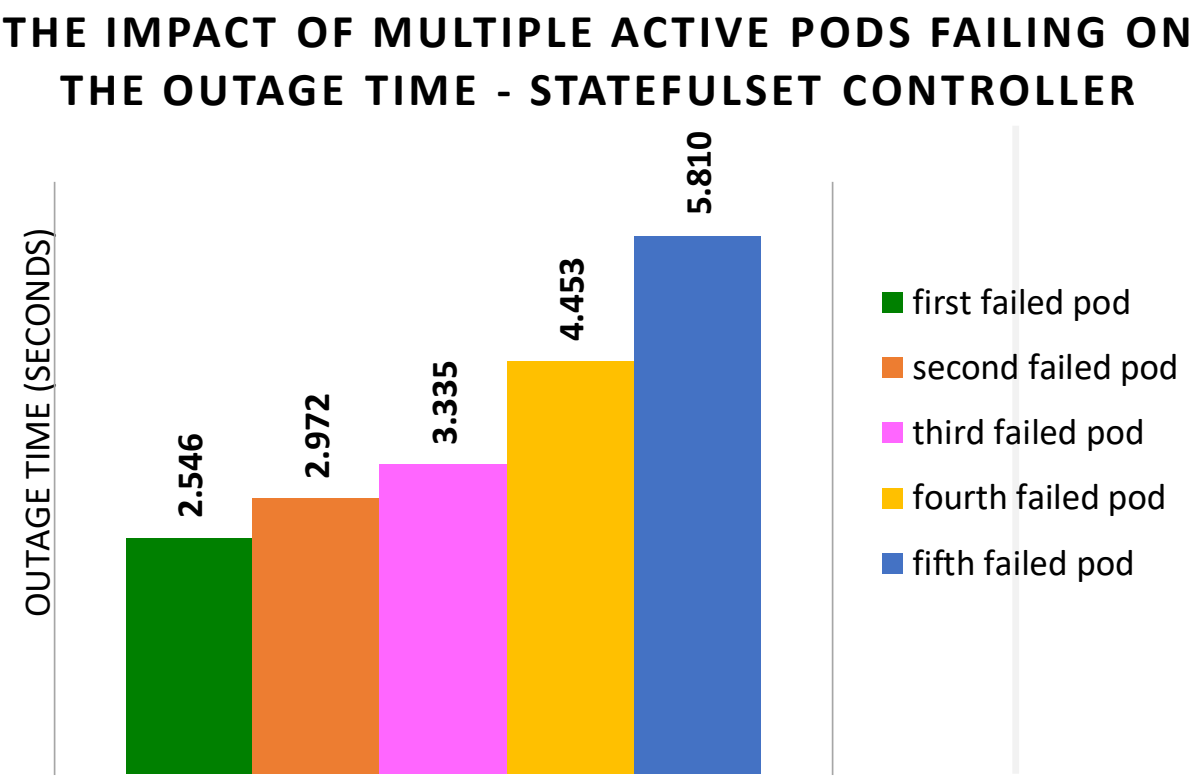

Figure 4-20. Results for experiments of RQ12 – StatefulSet controller (average outage time for each failed pod).

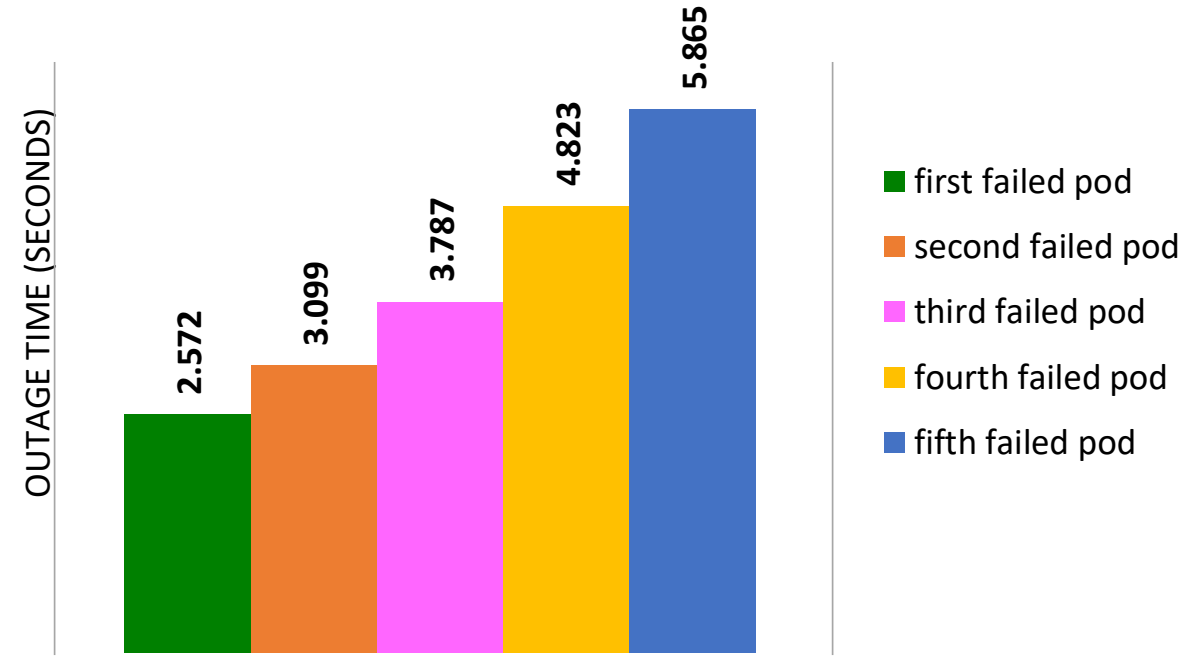

Figure 4-21. Results for experiments of RQ12 – Deployment controller (average outage time for each failed pod).

4.3 Conclusion

In this chapter, we proposed a solution to address the identified challenges of Kubernetes in providing availability for stateful microservice based applications and improve the availability. Our solution which is a State Controller allows for the automatic redirection of services to healthy pods through the management of secondary labels reflecting the current role of pods in the configuration from an availability perspective. Our solution allows for failure handling at the platform (i.e., Kubernetes) level and thus it closes a gap in Kubernetes when it comes to stateful microservice based applications. That is, in case of failure of the pod providing the service, the service is redirected to the healthy standby pod which is aware of the failed active pod's state. Therefore, it is capable of resuming its service. We observed that this redirection time may be significantly shorter than the restart of the failed pod of a StatefulSet controller.

In case of application container failure, with our solution, recovery happens before repair. Thus, we are able to improve recovery time by 55%. Moreover, applications deployed by StatefulSet controllers cannot recover from node failure if the node does not rejoin the cluster. Without our solution, recovery depends on how fast the node can reboot or rejoin the cluster. In such scenarios, we are able to improve recovery time by 99%. Since the State Controller communicates with the Kubernetes' API server, it can be easily integrated with Kubernetes and can work hand in hand with the current controllers in Kubernetes' binary.

Moreover, we enriched our proposed State Controller so it can provide availability for stateful microservice based applications whose number of microservice instances increase or decrease. With the modified State Controller, there can be multiple active and standby pairs. Our evaluations of the modified State Controller shows that integrating our solution improves

service recovery by 50% on average. However, we observed that when a scaling event happens during a failover process is being done by the modified State Controller, the outage time is increased by 16% and HA state assignment time for the scale-out scenario is increased by 48%. Moreover, we measured the scaling overhead of integrating the State Controller with Kubernetes between 7.5% and 10.5%. We also observed that that scaling and HA state assignment is done faster when the application is deployed by a Deployment controller compared to when it is deployed by a StatefulSet controller. Because unlike StatefulSet controllers, the Deployment controller does not add or delete pods in an order and one by one. While the fast deployment and HA state assignment of pods can be considered as a reason to deploy the application by a Deployment controller, one should consider the drawback of deploying stateful applications with Deployment controllers as well. That is, with Deployment controllers, service recovery is not guaranteed in the scale-in scenarios. Because Deployment controllers do not scale-in the application in a guaranteed manner and the standby of an active pod might be deleted in the scale-in process while the active pod remains in the pods' list. We also evaluated the availability provided by the modified State Controller when multiple active pods fail simultaneously and observed that the later the failure of a pod is detected by the State Controller, the longer its recovery time will be.

Finally, we acknowledge the threats to the internal, external, and construct validity of our results. One threat to the internal validity of our results is that the cluster for our experiments consists of a small number of nodes. Especially for the most responsive configuration, a larger Kubernetes cluster may add overhead and delays when detecting node failure which will certainly impact service availability. Another threat is the events that we consider as reaction, repair, or recovery time may be mapped differently. However, the change in the mapping will not affect the total outage time measured. The threat to the external validity of our results is

that we only considered the case of an on-demand video streaming application while other types of applications should be considered before generalizing the results. Moreover, the tools and mechanisms used in our experiments can be considered as threats to the construct validity of our results. While we used NTP for time synchronization, other methods such as container instrumentation may be used to be more precise.

The contents provided in this chapter are published in [52].

Chapter 5

Conclusion

In this thesis, we identified possible architectures for deploying stateless and stateful microservice based applications with Kubernetes. We evaluated these architectures from the perspective of availability and identified the issues that Kubernetes has in managing the availability of stateful microservice based applications. We proposed a solution that easily integrates with Kubernetes and improves the availability of its managed stateful applications.

For stateless microservice based applications, we conducted availability experiments for different failure scenarios to evaluate the repair actions of Kubernetes for providing availability for its managed applications. The results showed that in the failure scenarios where service outage is due to external execution failure events, the outage times are significantly longer compared to failure scenarios where service outage is due to internal administrative operations. However, in practice, Kubernetes' support for availability is demonstrated through internal administrative operations which does not reflect the performance of Kubernetes when external execution failures happen. For example, in the scenario where the service outage is due to node shutdown, the default configuration of Kubernetes will result in service outage of around 5 minutes. That is, the total allowed downtime over one year for the systems with high availability requirements. We also investigated the impact of adding redundant microservice instances on the availability and observed that it decreases downtime significantly. Because service recovery does not depend on the repair of the failed microservice instance. For stateful

applications, however, it does not hold true. That is, adding redundancy does not improve the availability for stateful applications. The reason is that in the event that a microservice instance fails, the redundant microservice instances neither know about the failure nor have the state of the failed pod to continue the service. Therefore, service recovery depends on the repair of the failed microservice instance. The results of our experiments show that the repair time (especially in node reboot scenarios) can be too long which can significantly decrease service availability. Moreover, in the scenarios where service outage is due to node shutdown, the microservice instance that was hosted by the failed node will not be repaired by Kubernetes. Therefore, service recovery does not even happen.

To address these issues, we proposed a solution that is a State Controller that integrates with Kubernetes and works hand in hand with the existing Kubernetes controllers and improves the availability of its managed stateful applications. The State Controller improves the availability through automatic redirection of the service to healthy microservice instances by managing secondary labels assigned to microservices of the application. Our solution implements the 2N redundancy model and assigns active and standby HA states. The standby microservice keeps the state of the active one. The State Controller notifies the standby microservice instance when a failure happens to the active one and also assigns active HA state to the standby microservice. Thus, it can resume the service instead of the failed microservice instance which takes less time compared to the repair of the failed microservice instance. We also modified our solution to provide availability for the application when it is scaled in or scaled out by keeping multiple pairs of active-standbys.

The experiments' results show that integrating the State Controller with Kubernetes improves recovery time from 55% to 99%. The reason for this improvement is that service recovery no longer relies on the repair of the failed microservice instance. Moreover, for the

scenarios where service outage is due to node shutdown that the failed microservice is not repaired, service recovery is not possible without our solution and integrating the State Controller enables service recovery for this type of failure as well. Since our solution does not require any change to Kubernetes' source code, it can be easily used to provide a higher level of availability for their existing containerized microservice based applications.

We identify high resource utilization as a limitation to our solution which is related to the 2N redundancy model where each standby microservice keeps protects only one active microservice. As future work, this limitation can be addressed by implementing other redundancy models such as N-Way redundancy model in order to have one standby microservice for a number of active microservice instances.

Bibliography

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," p. 7.

[2] N. Dragoni *et al.*, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham: Springer International Publishing, 2017, pp. 195–216.

[3] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, Jan. 2015.

[4] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015.

[5] "Microservices," *martinfowler.com*. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 01-Oct-2018].

[6] M. Toeroe and F. Tam, *Service Availability: Principles and Practice*. John Wiley & Sons, 2012.

[7] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," *Journal of Network and Computer Applications*, vol. 60, pp. 54–67, Jan. 2016.

[8] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34.

[9] "Docker - Build, Ship, and Run Any App, Anywhere." [Online]. Available: https://www.docker.com/. [Accessed: 01-Oct-2018].

[10] "Kubernetes Documentation." [Online]. Available: https://kubernetes.io/docs/home/. [Accessed: 01-Oct-2018].

[11] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," *Software: Practice and Experience*, vol. 48, no. 11, pp. 2019–2042, 2018.

[12] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How To Make Your Application Scale," in *Perspectives of System Informatics*, 2018, pp. 95–104.

[13] "OpenSAF Foundation - Welcome to OpenSAF." [Online]. Available: http://opensaf.org/. [Accessed: 12-Oct-2018].

[14] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using Docker technology," in *SoutheastCon 2016*, 2016, pp. 1–5.

[15] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[16] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015.

[17] "Microservices Architecture: Advantages and Drawbacks," *Cloud Academy*, 18-Oct-2018. .

[18] S. Fu, J. Liu, X. Chu, and Y. Hu, "Toward a Standard Interface for Cloud Providers: The Container as the Narrow Waist," *IEEE Internet Computing*, vol. 20, no. 2, pp. 66–71, Mar. 2016.

[19] "Docker Hub." [Online]. Available: https://hub.docker.com/. [Accessed: 18-Mar-2019].

[20] *NGINX Ingress Controller for Kubernetes. Contribute to kubernetes/ingress-nginx development by creating an account on GitHub*. Kubernetes, 2018.
[21] Andjelko Iharos, "HAProxy Ingress Controller for Kubernetes," *HAProxy Technologies*, 12-Dec-2017. .
[22] M. Toeroe and F. Tam, *Service Availability: Principles and Practice*. John Wiley & Sons, 2012.
[23] K. E. Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S. N. Rai, "The optimal class size for object-oriented software," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 494–509, May 2002.
[24] M. Villamizar *et al.*, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590.
[25] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.
[26] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency Analysis of Provisioning Microservices," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2016, pp. 261–268.
[27] H. Kang, M. Le, and S. Tao, "Container and Microservice Driven Design for Cloud Infrastructure DevOps," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 202–211.
[28] 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, Feb. 2017.
[29] H. V. Netto, A. F. Luiz, M. Correia, L. de O. Rech, and C. P. Oliveira, "Koordinator: A Service Approach for Replicating Docker Containers in Kubernetes," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 00058–00063.
[30] T. Soenen, W. Tavernier, D. Colle, and M. Pickavet, "Optimising microservice-based reliable NFV management amp;amp; orchestration architectures," in *2017 9th International Workshop on Resilient Networks Design and Modeling (RNDM)*, 2017, pp. 1–7.
[31] "www.etsi.org - /deliver/etsi_gs/NFV-EVE/001_099/011/03.01.01_60/." [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-EVE/001_099/011/03.01.01_60/. [Accessed: 07-Jun-2019].
[32] M. Toeroe, N. Pawar, and F. Khendek, "Managing application level elasticity and availability," in *10th International Conference on Network and Service Management (CNSM) and Workshop*, 2014, pp. 348–351.
[33] *Swarm: a Docker-native clustering system. Contribute to docker/swarm development by creating an account on GitHub*. Docker, 2019.
[34] I. Eldridge and pwpadmin, "What Is Container Orchestration?," *New Relic Blog*, 17-Jul-2018. [Online]. Available: https://blog.newrelic.com/engineering/container-orchestration-explained/. [Accessed: 29-Mar-2019].
[35] "cri-o." [Online]. Available: https://cri-o.io/. [Accessed: 29-Mar-2019].
[36] "containerd – An industry-standard container runtime with an emphasis on simplicity, robustness and portability." [Online]. Available: https://containerd.io/. [Accessed: 29-Mar-2019].
[37] *The hypervisor-based container runtime for Kubernetes.: kubernetes/frakti*. Kubernetes, 2019.

[38] “Marathon: A container orchestration platform for Mesos and DC/OS.” [Online]. Available: https://mesosphere.github.io/marathon/. [Accessed: 02-Apr-2019].
[39] “Apache Mesos,” *Apache Mesos*. [Online]. Available: http://mesos.apache.org/. [Accessed: 02-Apr-2019].
[40] “The Definitive Platform for Modern Apps,” *DC/OS*. [Online]. Available: https://dcos.io/. [Accessed: 02-Apr-2019].
[41] “ntp.org: Home of the Network Time Protocol.” [Online]. Available: http://www.ntp.org/. [Accessed: 12-Oct-2018].
[42] “VLC: Official site - Free multimedia solutions for all OS! - VideoLAN.” [Online]. Available: https://www.videolan.org/index.html. [Accessed: 12-Oct-2018].
[43] “Run a Replicated Stateful Application.” [Online]. Available: https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/. [Accessed: 04-Jan-2019].
[44] “Integrating Open SAF High Availability Solution with Open Stack - IEEE Conference Publication.” [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7196529. [Accessed: 12-Oct-2018].
[45] A. Kanso, M. Toeroe, and F. Khendek, “Comparing redundancy models for high availability middleware,” *Computing*, vol. 96, no. 10, pp. 975–993, Oct. 2014.
[46] “Production-Grade Container Orchestration.” [Online]. Available: https://kubernetes.io/. [Accessed: 12-Oct-2018].
[47] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, “Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 970–973.
[48] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, “Kubernetes as an Availability Manager for Microservice Applications,” *arXiv:1901.04946 [cs]*, Jan. 2019.
[49] “The Go Programming Language.” [Online]. Available: https://golang.org/. [Accessed: 21-Jan-2019].
[50] *Go client for Kubernetes. Contribute to kubernetes/client-go development by creating an account on GitHub*. Kubernetes, 2019.
[51] “Welcome! - The Apache HTTP Server Project.” [Online]. Available: https://httpd.apache.org/. [Accessed: 24-Nov-2018].
[52] L. A. 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)*, 2019, pp. 176–185.