Self-Healing Microservices with Kubernetes

Self-Adaptation in Micro-Service Architectures with Kubernetes Seminar – Summer Term 2019

Sebastian Schmidl[1]

Abstract: Abstract goes here.

Keywords: Self-Adaptive Systems; Self-Healing; Microservices; Cloud Computing; Kubernetes; Decentralized; Distributed; Orchestration

1 Introduction

Cloud Computing has become the de-facto standard of deploying new scalable applications. Companies chose cloud over on-premise or self-hosted environments, because they can deploy their applications more flexible, with higher and dynamically scalable performance, and because prices are very competitive [To15]. However, present cloud environments have to deal with heterogeneous resources and an ever-increasing scale. With this growing complexity failures are more likely to occur and software engineers have to design applications with that in mind. This can be achieved via replication, containment, isolation, and monitoring paired with responsive actions to failures [Bo14].

One way to realize containment and isolation of software components in a scalable way are microservice architectures. In this approach, the software application is decomposed along business domain boundaries into small, lightweight, autonomous services. Each service runs as its own application decoupled from the other services and acts as a scaling unit. Services communicate through lightweight REST APIs or asynchronous message queues. Microservice architectures embrace failure. If a service relies on another services, it is aware that the other service may not be available or the connection may be slow and the service can deal with the failures [Ne15].

For those complex distributed software systems consisting of hundreds of microservices, deployment and management gets more complex as well. The service management can be simplified by executing the microservices isolated from each other in containers, such as Docker containers [Do19]. Container orchestration tools, such as Kubernetes [Ku19],

[1] Hasso Plattner Institut, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, sebastian.schmidl@student.hpi.de

can further be used to deploy, scale, and manage containerized distributed microservice applications.

The increasing complexity of modern software systems motivated the development of self-adaptive systems. Those systems introduce an autonomous behavior that takes decisions at runtime and manages the complex underlying software system. This allows the software systems to adapt to unpredictable system changes and changing environments. Self-adaptive systems combine four self properties, as defined by Ganek; Corbi [GC03]:

self-configuring The systems adapt automatically to dynamically changing environments ("on-the-fly").
self-healing The systems discover, diagnose, and react to failures reducing disruptions and enabling continuous availability.
self-optimization Systems efficiently maximize resource utilization.
self-protection Systems anticipate, detect, identify, and protect themselves from attacks.

This list has been continuously extended and the extended properties are now referred to as self-* properties [PD11].

One essential part of a self-adaptive systems are its self-healing capabilities, which we will focus on in this paper. Self-healing systems monitor the running application and try to keep the system in an healthy state to increase availability and maintain the essential functionality. Therefore, they have to be fault tolerant, mask temporary failures, and recover from them in a finite amount of time to reach the healthy system state again. In cloud environments, self-healing techniques are already used in the form of approaches and tools that try to achieve continuous availability for cloud services. Those tools detect disruptions, diagnose failures and recover from them by applying an appropriate strategy [PD11]. As an example, consider Apache Mesos [Th19] and Mesosphere [Me19]. They perform periodic health checks to ensure a leader is available in the cluster all the time. This means they can detect failing nodes and replace them automatically. Other tools are located in the infrastructure layer: Amazon Auto Scaling [Am19] and Google Cloud Managed Instance Groups [Go19] monitor VM instances and automatically recreate them when they crash or stop. In addition, both tools allow the user to specify application health checks. However, in this paper we will focus on Kubernetes [Ku19].

In this paper, we compare the solutions and approaches proposed in self-healing research literature with the way Kubernetes implements self-healing capabilities for microservices in the cloud. We show, which self-healing concepts are already implemented by Kubernetes and where there are still open issues and limitations.

The rest of this paper is structures as follows: Sect. 2 introduces the concept of self-healing systems, summarizes recent research literature in this area, and discusses self-healing challenges in the cloud environment. In Sect. 3, we quickly present base concepts and the architecture of Kubernetes. Using those base concepts, we can then explain, how Kubernetes

implements self-healing capabilities in Sect. 4. The discussion of benefits and limitations of Kubernetes' self-healing capabilities follows in Sect. 5. We conclude this paper in Sect. 6.

2 Self-Healing

Self-healing is an integral part of self-adaptive systems and the focus of this paper. It combines properties of (i) fault-tolerant systems, which handle transient failures and mask permanent ones to ensure system availability, (ii) self-stabilizing systems, which are non-fault masking and converge to the legal state in a finite amount of time, and (iii) survivable systems, which maintain essential services and recover non-essential ones after intrusions have been dealt with [PD11]. A widely-used definition for self-healing systems is from Ghosh et al. [Gh07]:

> The key focus [...] is that a self-healing system should recover from the abnormal (or "unhealthy") state and return to the normative ("healthy") state, and function as it was prior to disruption.

This definition is very broad, but one can argue that the key aspect of self-healing systems are recovery oriented functionalities that bring the system back to the healthy state, which neither sole fault-tolerant systems nor sole survivable systems encompass [PD11].

Like in an autonomous system, the main component in a self-healing system is the self-healing manager. It runs a control loop with three stages that is a reduced version of the autonomic control loop, also referred to as MAPE-K loop [05]. The self-healing loop consists of the following three main stages [PD11]:

Detect The self-healing manager filters the status information about the running system and reports suspicious events and detected degradations.

Diagnose The diagnosis stage performs root cause analysis on the received reports from the previous stage and calculates a recovery strategy.

Recover In the recovery phase the manager applies the strategies to the system while he avoids any unpredictable side effects.

These three stages reflect the definition of self-healing. Fig. 1 shows the mapping of those stages to the MAPE-K control loop. There are two different ways, how a software system can be equipped with the above mentioned self-healing capabilities:

The first approach is a software application with built-in self-healing logic. This means that the self-healing manager is within the application code and is able to access internal state and mechanisms. This can be an advantage as the application is a white box and the self-healing logic can use detailed status information and even domain knowledge for detection, analysis and recovery. At the other hand, this also means that the healing logic is

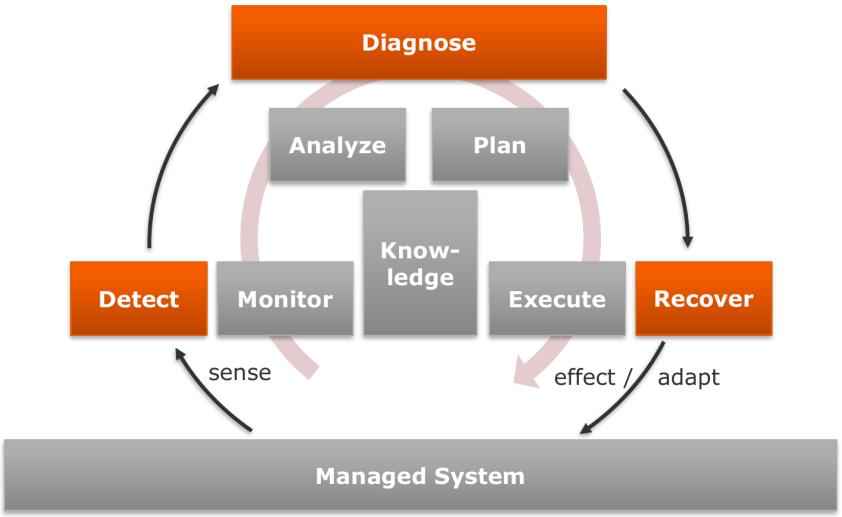

Fig. 1: Condensed autonomic control loop as the self-healing loop

tied to the application increasing coupling and violating the separation of concerns principle. If the application starves the self-healing manager may starve as well.

The second approach is an external self-healing manager provided as an infrastructural component or as third-party service. The self-healing logic runs in isolation from the application code and can therefore treat the application only as a black box and has to use external metrics to judge the application's health. It's the current state of the art for monitoring, health management and scaling logic [To15]. The external management logic has to be itself resilient, fault-tolerant, and scalable for being able to heal the application. Using third party services or services provided by the infrastructure provider could lead to vendor lock-in. There are also open-source alternatives that include self-healing capabilities and can be used as middleware between cloud infrastructure and the application, such as Kubernetes or Mesosphere.

2.1 Architecture-based self-healing

In the past most self-healing capabilities were included in the application code itself. This has the downside that the self-healing adaption engines and rules are tightly coupled to the application code and can't be reused. Researchers have therefore developed ways to generalize self-healing mechanisms. One approach are architecture-based self-healing systems, in which external self-healing managers perform repairs on the level of software components and connectors of the monitored systems [DHT02]. The assumption of those

approaches is that the component boundaries are the most loosely coupled points in the software system and can be reconfigured to allow changes in the architecture during runtime.

Dashofy et al. present the infrastructure to develop and run an architecture-based self-healing system that focuses on event-based software architectures in [DHT02]. They chose event-based architectures, because all included components can only communicate via events and show a significant degree of autonomy. This reduces coupling of the components and makes changes from the outside to the system possible.

2.2 Self-healing microservices

Microservice architectures take isolation of software components to the next level and improve the autonomy of the components (services). This makes applying repairs from the outside of the application easier, because services are decoupled, autonomous, and scalable and service boundaries are well-defined. External self-healing managers can monitor the microservices of the application and apply repairs to the failing or misbehaving ones by replacing the service with another one and rerouting the messages to the new instance. This is where the self-healing capabilities of the aforementioned cloud tools (refer to Sect. 1) and Kubernetes can be used.

In contrast to external self-healing managers, Toffetti et al. propose a self-healing system with the self-healing logic built into a microservice application. They leverage standard methods from distributed systems (i. e. consensus algorithms) to assign self-management (includes self-healing) functionality to some nodes of the distributed application. The selected nodes form a hierarchy and perform different parts of the self-management logic. This means that the self-management logic is distributed in the cluster to overcome the connectedness to the application code [To15].

2.3 Self-healing in cloud environments

As more and more applications are deployed in cloud environments, developers must consider the implications of running their software in the cloud. The Reactive Manifesto [Bo14] records the requirements, which today's software is facing. It asks for more resilient and responsive systems and postulates that resilience can be achieved by replication, containment, isolation, and delegation. Recovery should be handled by an external component to the application. This could be a self-healing manager. If the self-healing manager is run external to the application, it must itself be resilient to failures.

Yesterday's software was deployed on bare metal or virtual machines, but in the cloud multiple layers of abstractions and virtualizations are used. Most of the time this is transparent to the application as Platform as a Service (PaaS) solutions are used. This means that compute units are much easier to scale, exchange and expand. On the other hand, self-healing

strategies in cloud environments differ significantly from traditional ones, because the self-healing manager has no control over the underlying infrastructure and must work on higher abstraction levels.

While systems and software components can fail in various ways and research has come up with general failure classifications and resolutions [PD11, Tab. 1] for any software systems, failures in cloud environments can be reduced to one failure class: a node or component is detected unreachable. Although an unreachable node can have different root causes on the infrastructure level, the impact on the system is the same. Because application developers have no control about the infrastructure in a cloud deployment, they have to find other ways besides repairing the infrastructure to heal from failures in the cloud. Node failures can be detected via heartbeat messages or latency metrics and a common recovery strategy is the redeployment of the software that was running on the node on another node.

Research approaches that deal with these challenges for self-healing systems in the cloud are Stack et al. [St17], who developed an hierarchical approach using a master-slave architecture to provide flexible and high available self-healing functionality in a cloud architecture, and Florio; Nitto [FN16], who have a similar approach to Kubernetes. They developed their own self-healing container orchestration tool, called Gru, that targets microservice architectures deployed in containers.

3 Kubernetes

Kubernetes is an open-source platform for automating and managing distributed software in the cloud. It heavily relies on container technology and supports declarative configuration of the managed containers. Kubernetes was developed by Google and is open-source software since 2014 [Ku19].

Kubernetes is build on top of existing PaaS solutions and consists of a master-slave architecture, which is depicted in Fig. 2, forming a cluster. Slave nodes, Kubernetes calls them only “nodes”, are responsible for executing and maintaining the actual application via an underlying container runtime. Most of the time Docker [Do19] is used. The nodes also run Kubernetes components that manage cluster networking (`kube-proxy`) and interact with the container runtime and the master to monitor and manage the pods of the local node (`kubelet`). Pods are the smallest deployment unit in Kubernetes and consist of the application container (or multiple), connected resources, such as storages or network addresses, and the configuration options. The container runtime, `kube-proxy` and `kubelet` form the Kubernetes runtime environment [Ku19].

The Kubernetes master components provide the cluster’s control plane. As shown in the upper third of Fig. 2, they typically run only on one node, which does not run application pods. However, the components can be executed on any node in the cluster. The master components include (i) the `kube-apiserver` that exposes the control options to the user and other software,

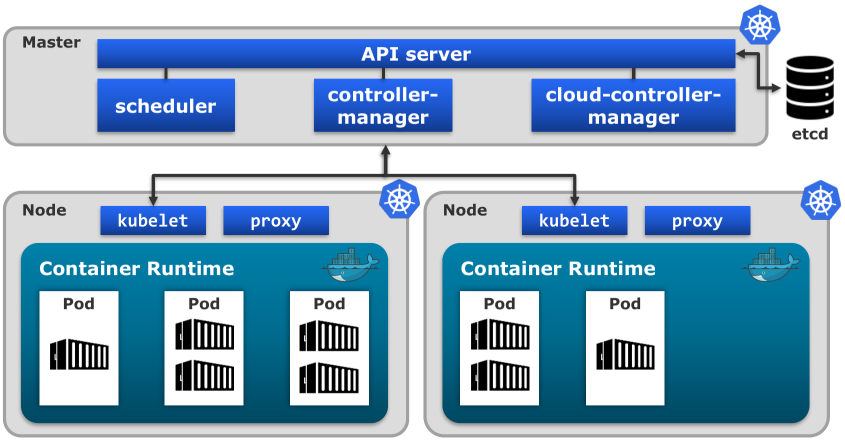

Fig. 2: Kubernetes architecture

(ii) an `etcd` [et19] instance as store for all cluster data, (iii) the `kube-scheduler`, which schedules newly created pods to the nodes in the cluster, (iv) the `kube-controller-manager`, which runs Node and Pod Controllers, and (v) the `cloud-controller-manager` to interact with the underlying cloud providers [Ku19].

For the declarative configuration and management of the application, Kubernetes employs the Kubernetes object model. All entities of the Kubernetes runtime are represented as description objects. The entirety of those objects represents the cluster state. Each object consists of three parts: (i) the object's metadata, such as name, version or labels, (ii) the object `spec`, which describes the desired state for the object and is provided by the user, and (iii) the object `status`, which describes the actual state of the object. Users of Kubernetes therefore only provide the object's metadata and `spec` to declare the desired deployment of their containerized applications on nodes and policies around how the application should behave. Kubernetes will constantly update the `state` of the objects according to the observed cluster state and takes corrective actions in the cluster to ensure that the cluster state matches the desired one declared in the objects' `spec` [Ku19].

4 Kubernetes' self-healing capabilities

In this section we will go through the self-healing capabilities available in Kubernetes. We will start with a short overview how Kubernetes approaches self-healing and what failure types exist in Sect. 4.1. We then explain, how Kubernetes implements the three properties of self-healing systems: fault-tolerant in Sect. 4.2, self-stabilizing in Sect. 4.3 and survivable in Sect. 4.4.

4.1 Overview

Kubernetes' self-healing capabilities are spread across various components and functionalities, but in essence they also perform the three-staged self-healing loop introduced in Sect. 2. Kubernetes' approach to self-healing is comparable to architecture-based self-healing [DHT02; To15]. Its declarative object configuration model resembles the concept of a desired and actual runtime architecture of the managed application. The user of Kubernetes can define the desired system architecture as the `spec` part of the deployment configuration. It is stored by the master components in `etcd`. Kubernetes then internally creates the `state` part of the objects in `etcd` by monitoring the actual nodes and pods in the cluster. This will detect failures in the cluster. The `states` represent the current architecture of the running components and are continuously updated. Based on those two representations corrective measures can then be calculated during the diagnosing stage. Kubernetes applies them to the cluster fully automatically to recover from failures. The actual state of the system thereby converges to the desired one.

There are two levels of disruptions that can occur in a Kubernetes deployment: Container failure and pod disruptions. Containers are runtime artifacts defined by users and can therefore fail or crash during execution. Those container failures are captured by the restart policy of their pod objects. When the restart policy is set to *Always* or *OnFailure*, failing containers are automatically restarted by the local `kubelet` component with an exponential back-off strategy. In this special case the self-healing loop is performed by the local `kubelet` component on each node.

In contrast to containers, pods do not disappear until someone (the user or a Kubernetes component) destroys them or there is an unavoidable system error. Kubernetes considers the following involuntary disruptions for pods [Ku19]:

- a hardware failure of the physical machine backing the node
- cloud provider or hypervisor failure makes VM disappear
- administrator deletes VM by mistake
- a kernel panic of the operating system
- a cluster network partition removes the node from the cluster

All those cases deal with failures of nodes or their communication. Therefore, Kubernetes employs *node-controllers* run by the *controller-manager* in the control plane (see Fig. 2) to monitor nodes. They detect node failures through a heartbeat-based failure detector and set the phase of all pods that where running on the failed node to *Failed*. This summarizes those failures into one category and propagates it to all the pods running on the node. This means the self-healing logic in Kubernetes only has to consider pod failures, thus we will only talk about healing from pod failures for the remainder of this section.

The following three sections explain how Kubernetes implements the three properties of self-healing systems introduced in Sect. 2 to heal from pod failures.

4.2 Fault-tolerant

Kubernetes uses pods and containers to isolate different parts of the managed application from one another. This fits well for microservice architectures and creates failure domains. Kubernetes can only provide fault-tolerance if the managed application makes use of pod replication and `Services`.

Pod replication means that for each microservice type, we let Kubernetes create multiple pods with the same configuration and containers. This means that we run multiple microservice instances at the same time. The `kube scheduler` automatically spreads those replicas across the available nodes to increase the failure tolerance`https://kubernetes.io/docs/concepts/configuration/assign-pod-node/`. This allows other replicas to take over the workload if one replica fails through an involuntary disruption.

To allow external services to communicate with our replicas, we must be able to reroute the traffic from failing pods to the remaining ones. This is done by specifying a `Service`, which exposes the set of replicated pods to the cluster internal and external network. It performs the load balancing and rerouting of the network traffic to the changing pod instances. This decouples the service access from the actual placement and deployment of the pods.

4.3 Self-stabilizing

Pods can be created manually, but this means the user has to take care of recovering failed pods. A better solution is to use Pod Controllers. They take pod description objects and manage their pods automatically. Pod Controllers execute the Detect – Analyze – Recover loop and run in the `kube-controller-manager` as a master component. Pod Controllers monitor the health of their managed pods via heartbeats and user-defined liveness probes[2]. Based on this monitoring the `state` part of the descriptor object is continuously updated to reflect the actual system state. Based on the desired state, the current system state, and the policies in the object, the Pod Controller derives corrective actions to transition the system from the current into the desired state. In the case of a failed pod, the controller would for example instruct the deletion of the failed pod and the creation of a new one with the same properties. The Pod Controller not only contains the self-healing logic for pods but also handles pod replication, rollout, and transparent pod placement on the available nodes. Using Pod Controllers, Kubernetes is able to recover failed pods and to eventually let the actual state converge to the desired one. This means applications managed by Kubernetes can be self-stabilizing.

There are four different types of pod payloads, which are unique to their failure handling: Stateless services, stateful services, daemons, and jobs. This is reflected in Kubernetes

[2] `https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/`

by different controller types. The next four sections cover the different aspects we have to consider for the failure handling of the application types.

4.3.1 Self-healing of stateless services

Stateless services as payloads for Kubernetes pods are easy to manage, because they can be destroyed and recreated on all nodes without any special resource dependencies. A stateless service can be defined via a `Deployment` or a `ReplicaSet`. In the case of a pod failure, those Pod Controllers instruct the deletion of the failed pod and the creation of a new one according to the supplied `spec`. The placement of the pods on the nodes is transparent and done by the `kube-scheduler`. The controllers always try to match the desired number of replicas.

4.3.2 Self-healing of stateful services

Stateful services can be defined via the `StatefulSet` deployment configuration. Pods managed by this type of Pod Controller have a unique identity including their name, network ID and configuration. They can be connected to `PersistentVolumes`, which are provided by the underlying infrastructure, such as a PaaS solution. `PersistentVolumes` are used as persistent storage for pods and once connected to a pod, they are also tied to the pods identity. If a pod fails, the controller reschedules it, potentially on another node, with the same identity. Therefore, the pod will reuse the assigned `PersistentVolumes` and network IDs. Routing of network traffic must be taken care of by creating a headless `Service`[3]. Stateful services using `PersistentVolumes` rely on the availability and fault-tolerance of the volumes provided by the underlying infrastructure.

4.3.3 Self-healing of daemons

Daemons are application components that should run once on all or selected nodes of the cluster, such as cluster storage, node monitoring, or a log collection component. Daemons can be defined via a `DaemonSet` deployment configuration. They ensure that a copy of the daemon runs on all the selected nodes. If a node with a daemon fails, no action is taken, as the desired state is still met, just with a reduced number of nodes. If a new node is added back to the cluster, the `DaemonSet` takes care of scheduling the creation of a new copy of the daemon on the newly added node. Recovering failed nodes must be done manually.

[3] `https://kubernetes.io/docs/concepts/services-networking/service/#headless-services`

4.3.4 Self-healing of jobs

Jobs only run once. They can be defined using the `Job` deployment configuration. The job Pod Controller ensures that the job is run to completion. This means if the job consists of one pod, this pod must terminate successfully, otherwise it will be restarted. Jobs can be started with parallel pods. In this case, the use can either specify a number of completions that must succeed or the controller waits for the first pod to successfully terminate. If those conditions are not yet met, the controller will recover failed pods.

4.4 Survivable

Survivable systems maintain the essential services of the managed application in the case of failure and resource pressure, while non-essential services may experience disruptions and are recovered after the failure has been dealt with. Kubernetes provides two ways for the user to influence the pod scheduling algorithm: `PriorityClasses` and *PodDisruptionBudget*s (PDBs). They can be used to compile a set of policies that help the scheduler to decide, which pods are essential and how many disruptions a set of replicated pods can handle.

`PriorityClasses` map to integer values, where a higher value indicates higher priority, and the user can assign them to the pods. Pod priority will affect the scheduling order of the pods. Higher priority pods will be scheduled first. In addition, under resource pressure higher priority pods in the scheduler queue will lead to lower priority pods being evicted by the scheduler. This is called preemption. If the user assigns high priority `PriorityClasses` to the essential services, the scheduler will do its best to recreate all instances of the essential services if they fail. It will even evict lower priority pods to make room for the essential services[4].

To prevent the scheduler to evict all instances of a lower priority service, the user can specify PDBs. They limit the number of replicated pods that are down due to voluntary disruptions, such as draining or preemption. Unfortunately, the scheduler only considers PDBs on best effort basis during preemption, which limits the applicability of PDBs for self-healing[5].

5 Discussion

As we have described in Sect. 4 Kubernetes provides all features required to setup a self-healing systems. Nevertheless, it requires the managed application to make use of the features and the user (developer or administrator) to configure the application deployment correctly. This includes packing the application parts in containers, writing an application that supports replication and distribution, using `Services`, Pod Controllers, `PriorityClasses`, and PDBs,

[4] `https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/`
[5] `https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#how-disruption-budgets-work`

and setting up external services for persistent storage of stateful services. Compared to implementing new recovery methods or specifying imperative recovery policies, Kubernetes is configured via the declarative definition of the desired system state. This provides a high abstraction and eases the usage of Kubernetes for equipping a system with self-healing capabilities in a cloud environment. The recovery and healing logic is provided by Kubernetes and thoroughly tested in big production environments. Kubernetes also provides a rich API to retrieve the current system state and update the desired state. This can be used to extend Kubernetes with further self-healing logic. Using Kubernetes for self-healing systems also comes with some limitations:

- Kubernetes has only an external view on the managed system. This is no problem at all, because failure recovery in the cloud domain is reduced to detecting nodes unreachable and re-deploying the software that was running on the node on another one (see Sect. 2.3).
- Kubernetes does not automatically repair failing infrastructure, such as nodes, external load balancers, or storage volumes. It relies on the availability and fault tolerance of the underlying infrastructure for `PersistentVolumes`.
- As Kubernetes is external to the application, it must itself be resilient and fault tolerant. Per default the Kubernetes' master components are only deployed on one node. This does not provide any fault-tolerance and therefore one must deploy a high-availability Kubernetes setup with multiple master nodes, potentially across availability zones. This setup is considerably complex and not yet automated.

6 Conclusion

In this work, we showed how Kubernetes can be used to equip a microservice application with self-healing capabilities. Kubernetes monitors the cluster state and takes corrective actions to let the cluster converge to the user-defined, desired state. Kubernetes implements fault-tolerance through the replication and isolation of application parts in pods and load balancing the network accesses to the pods using `Services`. On pod failures, other replicas can simply take over the failed node's work. After the failure, the cluster converges back to the desired state. Pod Controllers detect pod or node failures, diagnose them, and recover the failed pods by terminating the failed pod and scheduling a new identical one. This reflects the self-stabilizing aspect of self-healing systems. `PriorityClasses` and PDBs help the Kubernetes scheduler on resource pressure and occurring failures to maintain the essential services, while recovering the non-essential services when enough resources are available again. This allows the system to survive extreme stress situations. Kubernetes depends on the underlying infrastructure to provide fault-tolerant `PersistentVolumes` and load balancing and must be run in an high-availability setup to be itself resilient in the case of Kubernetes master component failures.

References

[05] An Architectural Blueprint for Autonomic Computing, tech. rep., IBM, 2005.

[Am19] Amazon Web Services, Inc.: AWS Auto Scaling, 2019, URL: https://aws.amazon.com/de/autoscaling/, visited on: 06/30/2019.

[Bo14] Bonér, J.; Farley, D.; Kuhn, R.; Thompson, M.: The Reactive Manifesto, 2014, URL: https://www.reactivemanifesto.org, visited on: 06/04/2019.

[DHT02] Dashofy, E. M.; van der Hoek, A.; Taylor, R. N.: Towards Architecture-based Self-healing Systems. In: Proceedings of the First Workshop on Self-healing Systems (WOSS). Pp. 21–26, 2002.

[Do19] Docker, Inc.: Docker – Enterprise Container Platform for High-Velocity Innovation, 2019, URL: https://www.docker.com/, visited on: 06/20/2019.

[et19] etcd Authors: etcd – A distributed, reliable key-value store for the most critical data of a distributed system, 2019, URL: https://etcd.io/, visited on: 06/22/2019.

[FN16] Florio, L.; Nitto, E. D.: Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures. In: IEEE International Conference on Autonomic Computing (ICAC). Pp. 357–362, 2016.

[GC03] Ganek, A. G.; Corbi, T. A.: The Dawning of the Autonomic Computing Era. IBM Systems Journal 42/1, pp. 5–18, 2003.

[Gh07] Ghosh, D.; Sharman, R.; Raghav Rao, H.; Upadhyaya, S.: Self-healing Systems - Survey and Synthesis. Decision Support Systems 42/4, pp. 2164–2185, 2007.

[Go19] Google, Inc.: Google Cloud Compute Engine Documentation – Instance Groups, 2019, URL: https://cloud.google.com/compute/docs/instance-groups/, visited on: 06/30/2019.

[Ku19] Kubernetes Authors: Kubernetes – Production-Grade Container Orchestration, 2019, URL: https://kubernetes.io/, visited on: 06/17/2019.

[Me19] Mesosphere, Inc.: Mesosphere – Focus on building apps, not inrastructure, 2019, URL: https://mesosphere.com/, visited on: 06/30/2019.

[Ne15] Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015.

[PD11] Psaier, H.; Dustdar, S.: A survey on self-healing systems: approaches and systems. Computing 91/1, pp. 43–73, 2011.

[St17] Stack, P.; Xiong, H.; Mersel, D.; Makhloufi, M.; Terpend, G.; Dong, D.: Self-Healing in a Decentralised Cloud Management System. In: Proceedings of the 1st International Workshop on Next Generation of Cloud Architectures. 3:1–3:6, 2017.

[Th19] The Apache Software Foundation: Apache Mesos – Program against your datacenter like it's a single pool of resources, 2019, URL: http://mesos.apache.org/, visited on: 06/30/2019.

[To15] Toffetti, G.; Brunner, S.; Blöchlinger, M.; Dudouet, F.; Edmonds, A.: An architecture for self-managing microservices. In: Proceedings of the 1st International Workshop on Automated Incident Management in Cloud. Pp. 19–24, 2015.