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 for the deployment, scaling, and management of the 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]. Kubernetes is one of those tools.

In this paper, we compare the solutions and approaches proposed in self-healing research literature with the way Kubernetes implements self-healing capabilities. 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 and summarizes recent research literature in this area. 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. Before we conclude the paper in Sect. 7, we briefly go through the related work in Sect. 6.

Where should I put this paragraph? While systems and software components can fail in various ways and research has come up with general failure classifications and resolutions [PD11, Tab. 1], failures in cloud environments can be reduced to one failure: 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 and we have no control about the infrastructure in a cloud deployment. This means that we have to find other ways besides repairing the infrastructure to heal from those failures. Node failures can be detected via heartbeat messages or latency metrics and a common recovery strategy used is the restart of the software that was running on the node on another node.

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. There are 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 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. Toffetti et al. propose such a system for a microservice architecture. 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.

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 themselves 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 can be used as middleware between cloud infrastructure and the application, such as Kubernetes.

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 the 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 Platform as a Service (PaaS) solutions and consists of a master-slave architecture 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. 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, (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 take 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. Kubernetes's 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, which can also be another software program as the Kubernetes API is machine-readable, 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 that represents the current architecture of the running components and updates them in `etcd`. Based on those two representations corrective measures can then be taken by Kubernetes fully automatically to let the actual state of the system converge 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 `kubelet` component with an exponential back-off strategy.

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

For all those cases Kubernetes sets the phase of all the pods that where running on the failed node to *Failed*, so the self-healing logic can take care of those failures.

Pods can be created manually, but this means one has to take care of pod failures as well. A better solution is to use pod controllers. They take pod description objects and manage the pods automatically. Pod controllers are the self-healing component for pods. They 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.

There are four different types of pod payloads, which require their own distinct failure handling, which is reflected in Kubernetes by different controller types: Stateless services, stateful services, daemons, and jobs. The next four sections cover the different aspects we have to consider for the failure handling of the application types.

4.1 Self-healing of stateless services

- Deployment definition via `Deployment` and the specification of replicas > 1 or with `ReplicaSet`
- Failing pods will be recreated to match the desired number of replicas (node placement is transparent)

4.2 Self-healing of stateful services

- Deployment definition via `StatefulSet`: `https://kubernetes.io/docs/tutorials/stateful-application/basic-stateful-set/`
- Uses `PersistentVolumes` (provided by the underlying cloud platform, e.g AWS, GCP, OpenStack) for storage
- Pods have a unique identity (name, network id, K8s configuration)

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

- Failed pods will be rescheduled on other nodes with their identity (re-using the assigned persistent volume and network id)
- A headless Service takes care of service discovery using SRV records and DNS (re-routing traffic to rescheduled pods on different nodes)
- therefore, relies on the availability and fault-tolerance of the used persistent volumes

4.3 Self-healing of daemons

- Defined via `DaemonSets`: `https://kubernetes.io/docs/concepts/workloads/controllers/daemonset`
- Ensures (monitors, restarts) that a copy of an application is run on each node (also on added or removed nodes)
- no real recovery if a node fails. Relies on manual action to replace the failed node. Then the `DaemonSet` will take care of creating the daemon pod on the newly added node.

4.4 Self-healing of jobs

- one-time jobs, terminating application

1. **the three "self-healing components":**
2. better put them in the discussion chapter? or in conclusion?
3. fault-tolerant through replication and isolation to ensure system availability:
    - Kubernetes is a distributed system, only one master per default, but master components can be placed on all nodes
    - Kubernetes uses pods and containers for isolation and containment
    - to ensure application availability: application must make use of provided features and support distributed deployment (especially replication)
4. self-stabilizing through pod controllers that continuously monitor system state and take actions to transition to desired state
5. survivable through: user can assign priority classes to critical pods and is able to create pod disruption budgets that limit the number of replicated pods that can are evicted simultaneously
    - `https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/`
    - we can define priority classes and assign pods to those
    - pod priority will affect scheduling order (higher priority pods first)
    - under resource pressure, higher priority nodes that are created and scheduled will evict lower priority pods (with their graceful termination period after which they are killed)

- pod disruption budgets can be specified to limit the number of replicated pods that are simultaneously down from voluntary disruption (draining, and also preemption)[3]
- pod disruption budgets are considered only on best effort basis during preemption

5 Discussion

1. requires containerized microservice application
2. code must support scaling and dynamic communication
3. provider of `PersistentVolumes` must ensure their availability and fault-tolerance
4. to deal with a node failure, remaining nodes must have enough spare capacity to host the failed pods or eviction kicks in
5. with replication factor 1, there are down times during re-creation of the pod on another node
6. limitations
    - **external management logic has to be themselves resilient, fault-tolerant, and scalable**
    - Kubernetes default only one master → HA setup across availability zones
    - quite a lot of configuration work, not automation yet (WIP)
    - only one master will be active (the other two will be passive), full state replication via etcd
    - fail-over will be handled by load balancer component
    - **only external view on the system**
    - **Kubernetes does not automatically repair or restart failing nodes**
    - –> automatic node repairs on GCE: `https://cloud.google.com/kubernetes-engine/docs/how-to/node-auto-repair`
    - components external to Kubernetes are not included in self-healing logic (such as external storage or load balancers of cloud provider)
7. benefits
    - healing from pod / container failures and node failures out-of-the-box
    - declarative definition of system state
    - rich API to retrieve current system state
8. interesting facts and insights
    - nothing not mentioned yet

6 Related Work

- **Is this chapter needed?** I could put a quick rundown of the approaches ([DHT02; FN16; To15]) into Sect. 2

[3] `https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#how-disruption-budgets-work`

- Reactive Manifesto [Bo14] asks for more resilient and responsive systems. The resilience is achieved by replication, containment, isolation, and delegation. Recovery should be handled by an external component. This could be a self-healing component.
- [St17]
  - hierarchical approach
  - layered master-slave architecture to provide flexibility and high availability
  - targets decentralized cloud architectures
- [To15]
  - architecture-based approach
  - self-managing microservice applications
  - within application
  - targets microservice applications
- [DHT02]
  - architecture-based approach
  - → repairs are done on the level of software components or connectors
  - external to application
  - targets event-based software architectures
- This approach
  - microservice container orchestrator approach
  - → repairs are done on container-level
  - external to application (Kubernetes)
  - targets microservice architectures deployed in cloud environments using containers and Kubernetes
- [FN16]
  - similar to this approach, but they have developed their own container orchestration tool, called Gru, instead of using Kubernetes
  - also targets microservice architectures deployed in containers

7 Conclusion

- self-healing in Kubernetes is an architectural approach
- achieves fault-tolerance through replication and redundancy
- on failure: redundant components take over
- after failure: the system converges to the desired state by rescheduling pods (pod controller)
- this reflects the self-stabilizing aspect of self-healing systems
- pod priorities and pod disruption budges help on resource pressure and failure to keep essential services running (through terminating non-essential ones and restarting them when more resources get available) to allow survivability of the system

References

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

[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.

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

[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.

[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.