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].

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].

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, such as Kubernetes. Those tools detect disruptions, diagnose failures and recover from them by applying an appropriate strategy [PD11]. 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 is 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.

Contribution and Summary

2 Related 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.
- [To15]
- [St17]
- [FN16]
- [DHT02]
- Kubernetes and alternatives

3 Self-Healing

1. sub control loop of MAPE-K loop (Detect – Analyze – Recover) [PD11]
2. different levels of self-healing (architecture-based, model-based, hierarchical, etc.)
3. self-healing management logic external and internal to the managed application
   external to application

   - self-healing and management logic is run in isolation from the application code
   - Examples: using services from the infrastructure provider, using third party services, or building an ad-hoc solution (e. g. using Kubernetes) [To15]
   - current state of the art for monitoring, health management, and scaling logic
   - could lead to vendor lock-in
   - external management logic has to be themselves resilient, fault-tolerant, and scalable

   within application

   - approach by Toffetti et al. for microservices; leverages standard methods from distributed systems (such as consensus algorithms) to assign self-management functionality to nodes of the application; hierarchical approach [To15]

4 Kubernetes

1. what is Kubernetes? –> `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`
2. architecture and how it works
   - master-slave architecture
   - master runs kube controller manager, API server, etcd, kube scheduler, cloud controller manager
   - slave (nodes) run kubelet (pod management and health monitoring) and kube proxy (cluster networking), and container runtime (e.g. Docker)
   - only slaves run application code
3. Kubernetes objects[2] and labels[3]
4. pods and containers[4]

5 Using Kubernetes to implement a self-healing application

1. How would a setup of a self-healing microservice architecture look like?
2. comparable to architecture-based approach
   a) Kubernetes object configuration corresponds to the desired runtime architecture of the managed application. ([To15] call it *instance graph*)
   b) Kubernetes internally holds the current architecture of the running components (in *etcd*)
   c) Container failures are captured by the restart policy of their pods. When set to *Always* or *OnFailure*, failing containers are restarted with an exponential back-off delay[5].
   d) To deal with node failures, pods have to be managed by controllers (explained later)[6]. They perform the Detect – Analyze – Recover loop by
      - monitoring the health of their managed pods with heartbeats and user-defined liveness probes[7]
      - comparing the desired and current state of their pods
      - performing actions (create or delete pod) to transition into the desired state
   e) Kubernetes sets the phase of all pods on a died or disconnected node to *Failed*
3. self-healing properties available in Kubernetes via controllers:
   "A Controller can create and manage multiple Pods for you, handling replication and rollout, and providing self-healing capabilities at cluster scope.

[2] `https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/`
[3] `https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/`
[4] `https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/`
[5] `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy`
[6] `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-lifetime`
[7] `https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/`

– https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pods-and-controllers" Pods are transparently placed on the available nodes by the controller.
    - recovery of stateful applications:
        - 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)
        - 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
    - recovery of stateless applications:
        - 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)
    - daemons: applications per node
        - 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. regarding the survivability aspect of self-healing systems: 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)[8]
    - pod disruption budgets are considered only on best effort basis during preemption

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

6 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
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

7 Conclusion

- short summary (microservices, self-healing, how Kubernetes does it)
- 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)
- 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)

References

[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.

[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.