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

1. cloud
    - cloud computing de-facto standard in industry
    - reasons: more flexibility, higher and dynamically available performance, and competitive prices [To15]
    - more hardware means more hardware can fail –> plan for failure [Ne15]
    - need for resilient systems [Bo14]
    - achieve it via replication, containment, isolation, and monitoring paired with responsive actions to failures
2. microservices
    - way to allow scaling of applications combined with a way to realize containment and isolation on business boundaries
    - decompose software application into small, lightweight, autonomous services (= scaling units)
    - embrace failure: services relying on other services should deal with them failing [Ne15]
3. deployment and orchestration → Kubernetes
4. self-adaptive systems
    - self-* properties
    - MAPE-K loop
5. self-healing

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

- currently 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.

- Neither fault-tolerant systems, nor survivable systems include recovery oriented functionalities that bring the system back to the healthy state, which is the key aspect of self-healing systems [Gh07].
- Combination of [PD11]
  - Fault-tolerant (handle transient failures and mask permanent ones)
  - self-stabilizing (non-fault masking; system converges to legal state in finite time and tries to remain in the same (closure))
  - survivable (maintain essential service and recover non-essential after intrusions have been dealt with)

6. self-healing of microservices in cloud environments
   - Psaier; Dustdar compare self-healing in cloud environments to the techniques that achieve continuous availability of the application [PD11]:

     > In cloud environments, self-healing can be considered as the techniques to achieve continuous availability, which involves detecting disruptions, diagnosing failures and recovering with a sound strategy.

   - In a cloud environment and a VM or container deployment, all failures are reduced to a single one: service unavailable after a while

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*)

[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/`

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. – `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)

[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/`

  - 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

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

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

  - 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.

[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.

[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.

[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.