Probleme und Erhöhung der Ausfallersicherheit von zustandsbehafteten Micrsoervices in Kubernetes

Michael Müller
michael.mueller2@haw-hamburg.de
Hochschule für angewandte Wissenschaften Hamburg
Department Informatik
20099 Hamburg, Germany

20. Februar 2019

Zusammenfassung

Um die Fehlertoleranz von zustandsbehafteten Microservices zu steigern, werden in dieser Ausarbeitung Use Cases definiert und Probleme bezüglich der Ausfallsicherheit analysiert. Zur Ausfallerkennung werden Heartbeats und zu Wiederherstellung werden Checkpoints verwendet.

Schlüsselwörter: Verteilte Systeme, Microservices, Ausfall, Heartbeats, Wiederherstellung, Checkpoints

1 Einleitung

Microservice Architekturen gewannen in den letzten Jahren an Zuspruch. Gründe hierfür sind die Erwartungen bezüglich einer gezielteren Skalierung, einer erhöhten Ausfallsicherheit, einer besseren Anpassung an neue Anforderungen und eine erhöhte Ausfallsicherheit ([21], [4], [7], [17], [9], [5]). Wie die Ausfallsicherheit für zustandsbehaftete Microservices in einem Kubernetes Cluster erhöht werden kann, behandelt diese Ausarbeitung im Rahmen des Grundseminars.

Ein Ausfall von zustandsbehafteten Microservices ist in Kubernetes ein Problem, da Kubernetes nicht genau erkennt, warum ein Microservice ausgefallen ist [13] und dessen Zustand nur unzureichend wiederherstellen kann [15].

In verteilten Systemen ist die Verwendung von Checkpoints zur Zustandswiederherstellung ([8], [18]) und die Ausfallerkennung anhand von Heartbeats ([2], [6], [11]) schon intensiv studiert worden. Diese Ausarbeitung thematisiert, welche Probleme angegangen werden müssen damit ein Ausfall erkennt werden kann und zeigt grob auf, wie ein verlorener Zustand wiederhergestellt werden kann.

2 Fehlererkennung

Die folgenden Alternativen beschreiben, wie ein Ausfall eines Prozesses (Fehler) erkannt werden kann (Fehlererkennung).

2.1 Heartbeats

Regelmäßig werden kleine Nachrichten (Heartbeats) von dem zu überwachenden Prozess an den Fehlerdetektor gesendet [1]. Die Heartbeats werden in Kombination mit Timeouts verwendet, damit ein Ausfall erkannt wird, wenn z. B. mehrere Heartbeats fehlen. Diese

Fehlererkennung wird zum Beispiel in [2], [6] und [11] verwendet.

2.2 Hardware Aktivität

[23] beschreibt eine Fehlererkennung, die den zu überwachenden Prozess in zwei Phasen aufteilt, Berechnungsphase und Kommunikationsphase. Während in der Berechnungsphase v.a. die CPU-Last steigt, steigt in der Kommunikationsphase die Last für die Festplatten, das Netzwerkes oder den CPU Kernel Speicher. Ist bekannt in welchem Zustand sich der Prozess befindet kann daraus abgeleitet werden, welche Ressource beansprucht wird. Ob der Prozess noch Aktiv ist, wird dann anhand der Auslastung der, für den Zustand relevanten, Ressource(n) ermittelt.

2.3 Fehlerdetektoren ohne Timeouts

Die meisten Fehlerdetektoren arbeiten mit Timeouts, um Ausfälle zu erkennen [3], jedoch gibt es gute Gründe dies nicht zu tun. Zwar nennen [1] und [16] Fehlerdetektoren die Timeouts nutzen, um zu bestimmen, ob ein Prozess ausgefallen ist, verwenden sie selbst allerdings nicht. [16] geht sehr stark darauf ein, dass ein Timeout nicht richtig gesetzt werden kann. Entweder der Timeout ist zu groß und es entsteht Wartezeit, bis der Ausfall erkannt wird, oder der Timeout ist zu klein und es werden "false positive" Ausfälle erkannt.

Das FALCON System von [16] benutzt keine Timeouts für die Ausfallerkennung, ist aber dafür wesentlich komplizierter. Das Falcon System bedingt, dass auf jedem Rechner mindestens ein kleines Überwachungsprogramm vorhanden ist das als "Spy" bezeichnet wird. Es ist pro Programm und Gerät bis zu ein Spy pro Ebene im Einsatz. Ebenen sind z. B. innerhalb des Prozesses, des Docker Containers, des Kubernetes Pods, des Kubernetes Nodes oder im Router. Mithilfe dieser kleinen Überwachungsprogramme kann der Status des zu beobachtenden Prozesses auf verschiedenen Ebenen abgefragt werden.

3 Wiederherstellung

Bei zustandsbehafteten Prozessen, muss nach Ausfall, der Zustand vor dem Ausfall wiederhergestellt werden. Wie eine solche Wiederherstellung umgesetzt werden kann, behandeln die folgenden Unterkapitel.

3.1 Protokollierung

Bei der Protokollierung wird vorausgesetzt, dass das Protokollierte erneut durchgeführt werden kann, dies sind zum Beispiel Transaktionen. Sobald eine Transaktion empfangen und abgeschlossen wurde, wird sie protokolliert. Fällt der protokollierte Prozess aus, so werden nach Neustart alle Transaktionen in derselben Reihenfolge erneut durchgeführt. Sind alle Transaktionen abgearbeitet, wurde der Zustand vor dem Ausfall wiederhergestellt ([24], [18]).

3.2 Checkpoints

Checkpoints sind persistierte Zustände eines Programms oder Daten zu einem bestimmten Zeitpunkt. Nach [8] gibt es mehrere Checkpointarten, die darin unterscheiden wie fein die Elemente der Checkpoints sind und wann ein Checkpoint eines Elements erstellt wird.

Die einfachste und hier weiterhin verwendete Checkpointart ist der "Transaction Consistent Checkpoint (TCC)". Die Definition besagt, dass zum Zeitpunkt der Erstellung keine Transaktion bearbeitet werden darf. Jede begonnene Transaktion wird abgeschlossen und alle neuen müssen warten, bis der Checkpoint erstellt wurde. Eine Alternative Checkpointart sind "Fuzzy Checkpoints", diese sind darauf ausgelegt schnell einen Checkpoint zu erstellen,

ohne Transaktionen aufzuhalten. Die Checkpoints werden Fuzzy genannt, weil sie in der Regel keine Konsistenz garantieren, da mehrere Objekte zu unterschiedlichen Zeitpunkten gespeichert wurden. Der TCC ist zur Wiederherstellung besser geeignet, da er einen kompletten konsistenten Zustand hält. Ein Fuzzy Checkpoint dagegen braucht für die Wiederherstellung weitaus länger, denn hier werden gegebenenfalls noch weitere Protokoll Daten gebraucht um aus dem inkonsistenten Zustand in einen konsistenten zu gelangen.

Bei der Wiederherstellung verwenden Checkpoints in der Regel zusätzlich ein Protokoll der durchgeführten Transaktionen die zwischen der Checkpoint Erstellung und des Ausfalls bearbeitet wurden. In Kombination mit dem Protokoll gehen weniger bereits bearbeitete Transaktionen verloren ([8], [18]).

4 Ausfälle in verteilten Systemen

Ein Ausfall wird als das nicht Erfüllen von Zusagen definiert [24]. Ein Fehler ist Teil des Systemzustands, der zu einem Ausfall führen kann.

Für Prozessausfälle gibt es unter anderem zwei Abstraktionen, "crash-stop" und "crash-recover". Bei "crash-stop" wird davon ausgegangen, dass ein ausgefallener Prozess frühestens dann wiederhergestellt ist, wenn das Ziel durch die restlichen Prozesse schon erreicht wurde, zum Beispiel in einem verteilten Algorithmus. Mit eingeschlossen ist auch keine oder eine nicht erfolgreiche Wiederherstellung. Dagegen ist bei der "crash-recover" Abstraktion vorgesehen, dass ein Prozess ausfällt und daraufhin sofort wiederhergestellt wird [3]. In dieser Ausarbeitung wird die "crash-recover" Abstraktion verwendet.

5 Microservices

Microservices sind autonome Services die zusammen arbeiten [21]. Sie haben eine hohe Kohäsion und sind gegenüber anderen Microservices gering gekoppelt [20]. So führt ein Ausfall eines Microservices nicht zwingend zum Absturz des Gesamtsystems wie bei einem Monolithen.

Die Microservices kommunizieren in der Regel über das Netzwerk, da sie in der Regel auf unterschiedlichen Computern ausgeführt werden. Dies hat eine Erhöhung der Autonomie und Komplexität zur Folge. Letzteres äußert sich v. a. während des Debuggings [21].

Microservices werden als Verfeinerung der "Service Orientierten Architektur" angesehen und sind auch ein Architektur Pattern ([5], [10]).

Microservices können in zustandsbehaftete und zustandslose Microservices unterschieden werden. Der Unterschied von zustandsbehafteten Microservices zu zustandslosen Microservices ist der, dass zustandslose Microservices bei selben Input immer denselben Output erzeugen. Bei einem zustandsbehafteten Microservices hängt der Output vom dem Zustand ab, in dem der Microservices sich aktuell befindet [22].

Bei der Wiederherstellung, nach einem Ausfall, muss diese Eigenschaft eines Microservices berücksichtigt werden. Während ein zustandsloser Microservice ohne weiteres Neu gestartet werden kann, muss bei einem zustandsbehafteten Microservice der Zustand vor dem Ausfall wiederhergestellt werden. Wie in Kapitel 3 erwähnt, ist dies zum Beispiel mit Checkpoints möglich.

6 Kubernetes

[19] beschreibt Kubernetes als "Cluster Management Tool" und Hilfsmittel, das einzelnen

Ressourcen (Speicher, Computer) in ein Cluster zusammenfasst. Kubernetes übernimmt die Verwaltung von ausgeführten Programmen und lastet alle Computer (Nodes) gleichmäßig aus.

Kubernetes besteht aus Pods und Nodes. Ein Node ist zum Beispiel ein Computer auf dem Pods aktiv sein können. Innerhalb der Pods können wiederum Container gestartet werden. Kubernetes Services laufen in einem oder mehreren Kubernetes Pods, diese Pods sind die kleinste Einheit die zur Lastverteilung herangezogen werden kann [14].

7 Problem- und Anforderungsanalyse

In diesem Kapitel werden die Use Cases und Problemfälle beschrieben.

7.1 Seperater Service

Theoretisch besteht die Möglichkeit, dass der Prozess innerhalb des Microservices selbst einen Ausfall erkennt. Dies ist eine valide Idee, solang der Fehler innerhalb des Prozesses auftritt. Fällt jedoch z. B. der ganze Kubernetes Node oder Pod aus, so kann der Prozess dies nicht erkennen da er selbst nicht mehr existiert. Somit braucht es einen externen, in Redundanz vorhandenen, Microservice der die Microservices überwacht. Der Crash Detector Microservice hat die Aufgaben den anderen Microservice zu überwachen und die Wiederherstellung anzustoßen.

Im weiteren Verlauf der Arbeit ist mit "Microservice" nur noch der zu überwachende Microservice gemeint. Bezieht sich etwas auf den Crash Detector wird dieser namentlich genannt.

7.2 Use Cases

Die erforderte Funktionalität kann in fünf Use Cases eingeteilt werden.

UC01	Anmeldung des Microservices
UC02	Erstellung der Checkpoints
UC03	Senden von Heartbeats
UC04	Prüfen der Checkpoints, Heartbeats und ggf. Neustarten
UC05	Abmeldung des Microservices

Tabelle 1: Die festgestellten Use Cases

Folgend werden alle Use Cases einzeln näher erläutert.

7.2.1 UC01 Anmeldung

Damit der Crash Detector weiß, welche Microservices ausgefallen sind, ist es wichtig zu wissen, welche gestartet wurden. Aus diesem Grund meldet sich der Microservice bei dem Crash Detector, dies gilt auch bei Neustart nach einem Ausfall.

7.2.2 UC02 Checkpoint Erstellung

Die Erstellung der Checkpoints ist für jeden Microservice speziell anzupassen. Grundsätzlich bleibt bei der Implementation des Microservices überlassen, welche Checkpointart implementiert wird und wie bzw. wann die Checkpoint Erstellung bzw. Wiederherstellung stattfindet. Präferiert ist der erwähnte TCC Checkpoint (Kapitel 3).

Unabhängig von der speziellen Aufgabe des Microservices sendet der Microservice dem Crash Detector nach Erstellung eines Checkpoints die ID des Checkpoints zu. Nach Erhalt dieser ID wird der Crash Detector diese zu den gespeicherten Daten über den zu überwachenden Microservice hinzufügen.

7.2.3 UC03 Heartbeats

Durch die Anmeldung ist dem Crash Detector bekannt, welche Microservices aktiv sein sollten. Um festzustellen, ob ein Microservice noch

aktiv ist, werden Heartbeats verwendet (siehe Kapitel 2).

Die Heartbeats werden hierbei in einem spezifizierten Rhythmus von den Microservices an den Crash Detector geschickt. Der erste Heartbeat wird nach der Anmeldung bzw. der letzte Heartbeat vor der Abmeldung gesendet.

7.2.4 UC04 Prüfung & Neustart

Regelmäßig überprüft der Crash Detector die Checkpoints (IDs) und Heartbeats die er über die Microservices gespeichert hat. Es wird geprüft, ob die gespeicherten Checkpoints bestimmte Kriterien erfüllen, wenn nicht werden sie gelöscht. Die Heartbeats werden auf ihre Aktualität geprüft.

Sollte ein Ausfall erkannt worden sein, so wird der Microservice neu gestartet. Wie schon bei UC02 ist zu beachten, dass der Neustart und die damit verbundene Zustandswiederherstellung je nach Aufgabe des Microservice angepasst werden muss. Die Zustandswiederherstellung wird vom Microservice selbst übernommen, nicht vom Crash Detector, dieser startet nur neu.

7.2.5 UC05 Abmeldung

Hat der Microservice seine Aufgabe beendet oder muss aufgrund eines Fehlers seine Arbeit beenden, meldet er sich vom Crash Detector ab. Der Crash Detector wird alle Daten löschen, die weiterhin nicht von Relevanz sind, dies sind z.b. die gespeicherten Daten über die Heartbeats oder ggf. die Checkpoints.

7.3 Ausfälle

In diesem Kapitel wird auf die verschiedenen Ausfälle eingegangen.

7.3.1 Kubernetes Pod Ausfallerkennung

Kubernetes erkennt den Ausfall eines Docker Containers anhand seines Exit Codes oder durch Abfragen einer REST Schnittstelle [12]. Ist der Exit Code eines Containers nicht 0 wird der Pod als Ausfall deklariert [13]. Eine native Alternative besteht darin, eine REST Schnittstelle für Kubernetes anzubieten. Sobald diese Schnittstelle einen HTTP Code zurückgibt der nicht mehr zwischen 200 und 300 liegt oder gar keine Antwort empfangen wird, erkennt Kubernetes den Container als ausgefallen.

Beide Alternativen decken nicht den Fall ab, dass der Container des Microservices durch einen Laufzeitfehler beendet wurde. In diesem Fall ist der Exit Code nicht 0, da der Container aufgrund eines Fehlers terminierte und die REST Schnittstelle kann nicht mehr antworten, weil der Container schon terminiert ist. Ein Laufzeitfehler kann also in einem Kubernetes Cluster dazuführen, dass ein Container stetig terminiert und neu gestartet wird. Der Neustart durch Kubernetes lässt sich deaktivieren, dies muss allerdings explizit in den Kubernetes Deployments definiert werden.

7.3.2 Ausfall eines Microservices

Ein zu überwachender Microservices kann zu jeder Zeit ausfallen. Es wird in folgende Zustände unterschieden:

Zustand 1 Ausfall vor Anmeldung

Zustand 2 Ausfall nach Anmeldung

Zustand 3 Ausfall während der Erledigung der Aufgabe

Zustand 4 Ausfall vor Abmeldung

Zustand 5 Ausfall nach Abmeldung

Ausfälle in den Zuständen 1, 2, 4 und 5 sind die einfachen Fälle. Es ist nicht zu erwarten,

dass ein Ausfall in diesen Zuständen oft passieren wird, da sie nur sehr kurz sind. Ein Ausfall in Zustand 1 und 5 kann vom Crash Detector nicht erkannt werden, es muss manuell eingegriffen werden. Ausfälle in Zustände 2 und 4 dagegen werden vom Crash Detector erkannt und werden zu einem Neustart führen.

Die Ausfälle während Zustand 3 sind die relevanten. Ausfälle können hier zu Daten- oder Rechenverlust führen. In diesem Zustand wird die eigentliche ggf. zeitaufwendige Aufgabe abgearbeitet. Zudem muss der Microservice durch den Crash Detector neugestartet werden, ein Neustart führt dazu, dass der Microservice den aktuellsten Checkpoint läd und daraus den verlorenen Zustand wiederherstellt.

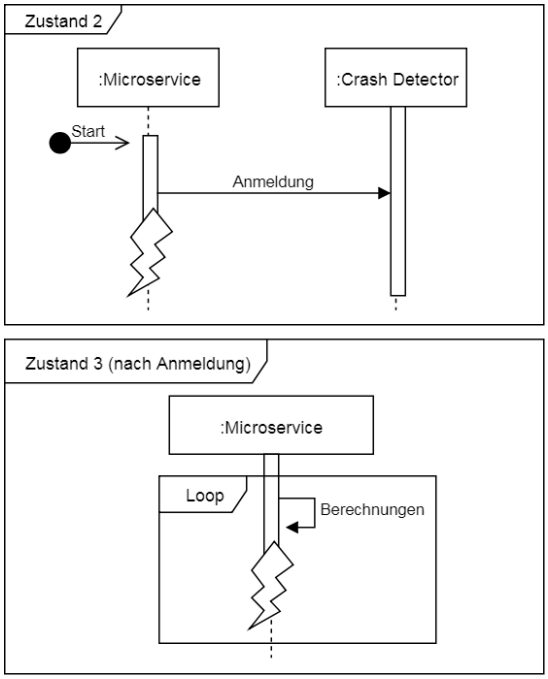

Abbildung 1: Ausfälle des zu überwachenden Microservices

7.3.3 Crash Detector Ausfall

Um die Ausfallsicherheit der Microservices zu gewährleisten ist es wichtig, dass der Crash Detector stets erreichbar ist. Dies ist z. B. über mehrere Crash Detectors zu lösen. Dann jedoch muss beachtet werden, dass auch ein Crash Detector ausfallen kann. Es gibt folgende Zustände, in denen ein Crash Detector ausfallen kann:

Zustand 1 Der Crash Detector bearbeitet keine Anfrage und prüft nichts.

Zustand 2 Der Crash Detector bearbeitet gerade eine Anfrage oder prüft.

Der 1. Zustand ist hierbei der leichteste, da der Crash Detector in dem Moment nichts tut, kann dieser Zustand sehr leicht wiederhergestellt werden. Für die Wiederherstellung muss nur ein neuer Crash Detector gestartet werden.

Bei Zustand 2 muss berücksichtigt werden, dass die Aktion (Bearbeitung einer Anfrage, Prüfung) erneut durchgeführt werden muss. Die Daten die während der fehlgeschlagenen Aktion manipuliert wurde, müssen wieder in den Ursprungszustand zurückgesetzt werden.

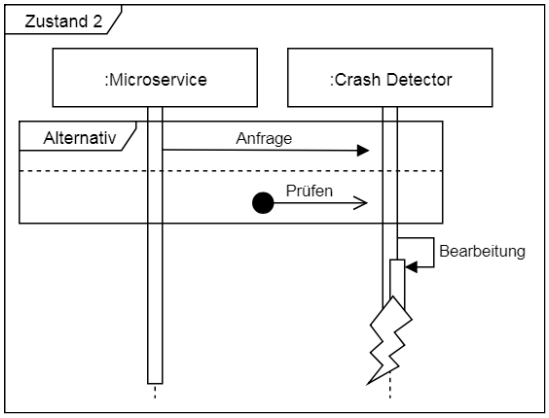

Abbildung 2: Ausfall in Zustand 2 des Crash Detectors

7.3.4 Verlorene Nachrichten

Wie in anderen verteilten Systemen, können hier auch Nachrichten verloren gehen. So werden hierbei folgende Fehlerfälle beachtet:

Ausfall 1 Die Nachricht von dem zu überwachenden Microservice zum Crash Detector

kam nicht an.

Ausfall 2 Die Antwort des Crash Detectors kam nicht bei dem zu überwachen Microservices an.

Mit Ausfall 1 muss so umgegangen werden, dass man nach endlicher Zeit die Nachricht erneut schickt oder ohne erneutes Senden fortfährt, sonst entsteht ein Deadlock.

Der 2. Ausfall ist dem ersten aus Sicht des Microservices sehr ähnlich, denn er bekommt keine Antwort. Allerdings nicht aus Sicht des Crash Detectors, denn dieser kann aufgrund der Nachricht schon Daten manipuliert haben. Der Ausfall wird toleriert, in dem der zu überwachende Microservice wie in Ausfall 1 reagiert. Ein erneutes Senden der Nachricht darf keine weitere Manipulation mit sich ziehen.

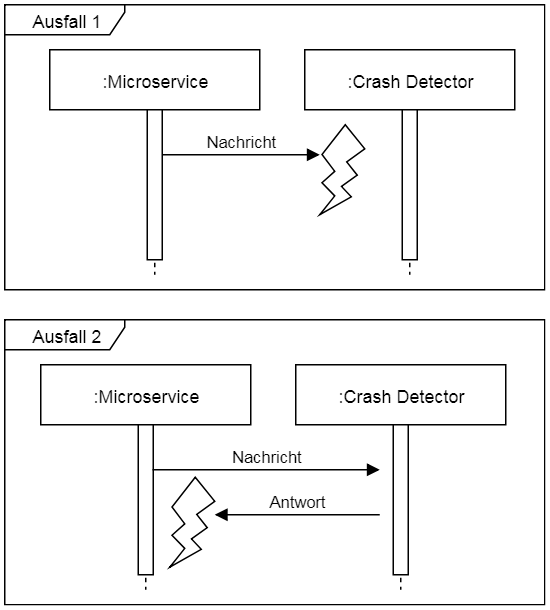

Abbildung 3: Fehler beim Senden von Nachrichten

7.3.5 Netzwerk Ausfall

Hin zu den bereits besprochenen Fehlern kann auch das Netzwerk instabil werden. Hierfür berücksichtigte ich die folgenden Fehler:

Ausfall 1 Der zu überwachende Microservice erreicht keinen Crash Detector.

Ausfall 2 Der zu überwachende Microservice ist vom Cluster abgeschnitten.

Ausfall 3 Der Crash Detector ist vom Cluster abgeschnitten.

Ausfall 4 Der Crash Detector und der zu überwachende Prozess sind vom Cluster getrennt können sich aber gegenseitig noch erreichen.

Ausfall 1 führt dazu, dass sich der zu überwachende Microservice nicht An- oder Abmelden kann und dem Crash Detector keine neuen Checkpoint IDs zu senden kann. Der Crash Detector wiederum muss richtig erkennen, warum die Heartbeats ausbleiben und darf keinen Neustart anstoßen. Siehe auch Kapitel 7.2.2, 7.2.3 und 7.2.4.

Ausfall 2 stellt den Totalausfall für den Microservice dar, basiert die zu erledigende Aufgabe des Microservices darauf über das Netzwerk zu kommunizieren, kommt die Aufgabe zu einem Halt. In diesem Fall muss von außen manuell eingegriffen werden. Analog zu Ausfall 2 ist Ausfall 3 der Totalausfall aus Sicht des Crash Detectors. Ausfall 4 ist ähnlich, zwar hat der Microservice noch Kontakt zu Crash Detector, kann aber nicht weiter mit anderen Systemen kommunizieren.

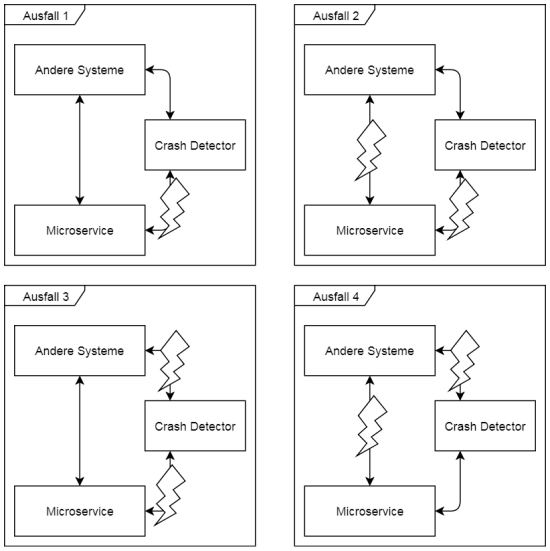

Abbildung 4: Fehler durch instabiles Netzwerk

8 Zusammenfassung

Mithilfe von Checkpoints und Heartbeats kann die Ausfallsicherheit von zustandsbehafteten Microservices erhöht werden. Aufgrund fehlender Funktionalität der Kubernetes eigenen Ausfallerkennung, muss ein neuer Microservice erstellt werden der die Überwachung übernimmt. Die Probleme für einen ausfallsicheren zustandsbehafteten Microservice wurden beschrieben, diese gilt es bei zukünftiger Arbeit zu lösen.

Diese Ausarbeitung kann in Zukunft vor allem in Richtung Struktur der Checkpoints erweitert werden. Dann gilt es zu klären, welche Daten ein Checkpoint halten muss, dass aus diesem der vollwertige Zustand wiederhergestellt werden kann.

Desweiteren kann überlegt werden wie die Crash Detectors zusammen arbeiten, so dass alle auf demselben Stand arbeiten. Erste Ideen könnten eine stetige Synchronisation untereinander oder eine gemeinsame Datenbank sein.

Literatur

[1] Marcos Kawazoe Aguilera, Wei Chen, Marcos Kawazoe, Aguilera Wei, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication, 1997.

[2] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[3] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer, 2011.

[4] Thomas F. Düllmann and André van Hoorn. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 171–172, New York, NY, USA, 2017. ACM.

[5] Martin Fowler. Microservices - a definition of this new architectural term, März 2014.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[7] David García Gil and Rubén Aguilera Díaz-Heredero. A microservices experience in the banking industry. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, ECSA '18, pages 13:1–13:2, New York, NY, USA, 2018. ACM.

[8] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[9] Wilhelm Hasselbring. Microservices for scalability: Keynote talk abstract. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 133–134, New York, NY, USA, 2016. ACM.

[10] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. Performance engineering for microservices: Research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 223–226. ACM, 2017.

[11] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[12] Kubernetes. Configure liveness and readiness probes, Juli 2018.

[13] Kubernetes. Pod lifecycle, December 2018.

[14] Kubernetes. Pod overview, November 2018.

[15] Kubernetes. Statefulsets, November 2018.

[16] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, New York, NY, USA, 2011. ACM.

[17] Qiankun Li, Gang Yin, Tao Wang, and Yue Yu. Building a cloud-ready program: A highly scalable implementation based on kubernetes. In *Proceedings of the 2Nd International Conference on Advances in Image Processing*, ICAIP '18, pages 159–164, New York, NY, USA, 2018. ACM.

[18] Jun-Lin Lin and Margaret H. Dunham. A survey of distributed database checkpointing. *Distrib. Parallel Databases*, 5(3):289–319, July 1997.

[19] Marko Luksa. *Kubernetes in Action*. Hanser Verlag, 2018.

[20] Manuel Mazzara and Bertrand Meyer. *Present and Ulterior Software Engineering*. Springer Publishing Company, 1st edition, 2017.

[21] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[22] Kristopher Sandoval. Defining stateful vs stateless web services, Mai 2017.

[23] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Light-weight black-box failure detection for distributed systems. In *Proceedings of the 2012 Workshop on Management of Big Data Systems*, MBDS '12, pages 13–18, New York, NY, USA, 2012. ACM.

[24] Andrew S. Tanenbaum and Maarten van Steen. *Verteilte Systeme - Prinzipien und Paradigmen*. Pearson Studium, 2008.