Resilience für Cloud Anwendungen

Dennis Pietruck *Department of Computer Science*
Hamburg University of Applied Sciences
Hamburg, Germany
Dennis.Pietruck@haw-hamburg.de

I. Einleitung

Mit der Verlagerung von Dienstleistungen auf Internetplattformen und daraus resultierenden Geschäftsmodellen ist es für Plattformanbieter von besonderer Bedeutung die Erreichbarkeit sowie die Performanz der angebotenen Dienste sicherzustellen. Für einen Onlineshop zum Beispiel bedeuten Ausfälle Zeit in der keine Geschäfte abgewickelt werden können und damit einen finanziellen Schaden. Weiterhin bewirken Ausfälle, sowie für den Benutzer wahrnehmbare Latenz eine Verschlechterung des Nutzererlebnis was zu einer Minderung der Nutzerzahlen führen kann.

Der Microservice Architekturstil ist eine Architektur bei der ein System aus einzelnen unabhängigen Anwendungen, die eine spezifische Aufgabe erfüllen sollen, zusammengesetzt wird. Dabei kommunizieren die einzelnen Komponenten vor allem über das Netzwerk, da sie verteilt auf mehreren Rechnern ausgeführt werden können. Vor allem bei Plattformen die mit einer großen Nutzerzahl konfrontiert sind wird diese Architektur häufig verwendet, da die Funktionalitäten der Gesamtanwendung individuell skaliert werden können. Gegenüber dem Skalieren einer einzelnen monolithischen Anwendung ist diese Vorgehensweise effizienter.

Wie auch in monolithischen Systemen können Ausfälle durch Programmierfehler oder durch hohe Last aus vielen Nutzeranfragen verursacht werden. Allerdings entstehen durch eine verteilte Architektur weitere Schwachstellen. Dadurch, dass das System auf mehreren Rechnern ausgeführt wird, können auch potentiell mehrere Komponenten des Systems ausfallen. Eine weitere Schwachstelle bildet das Netzwerk über das die einzelnen Komponenten kommunizieren. Verbindungsabbrüche führen dazu, dass wichtige Dienste im Netzwerk nicht erreicht werden können. Auch hohe Latenzen können starke Last für eine verteilte Anwendung erzeugen, was einen Systemausfall verursachen kann, oder sich als erhöhte Wartezeit für Endanwender bemerkbar macht, was zu einer Minderung des Nutzererlebnis führt.

Um verteilte Anwendungen ausfallsicherer zu machen wurden Strategien und Frameworks entwickelt. Dazu gehören Patterns die verhindern sollen, dass Ausfälle in einer Komponente zu Ausfällen in weiteren Komponenten führen und Patterns die das Systems vor hoher Last durch viele Benutzer schützen sollen. Außerdem wurden Methoden entwickelt mit denen Schwachstellen erkannt werden sollen, bevor sie Fehler in der gesamten Anwendung verursachen. In dieser Arbeit soll ein Überblick über Methoden gegeben werden mit denen die Widerstandsfähigkeit von Anwendungen gestärkt werden kann.

II. Widerstandsfähigkeit durch Pattern

Wenn eine Softwarekomponente nicht mehr erreichbar ist, kann es viele Gründe dafür geben. Dazu gehören Netzwerkpartitionierung oder Ausfall der Maschine auf der die Komponente ausgeführt wird. Außerdem kann der Ausfall der Komponente durch Programmierfehler verursacht worden sein. Unittests dienen dazu diese zu finden bevor eine Anwendung im Produktivsystem ausgeführt wird. Dennoch werden in Softwareprojekten selten alle defekten Codestellen gefunden [1]. Deshalb werden Pattern verwendet um die Auswirkung eines Fehlers einzuschränken.

Im Folgenden werden Pattern vorgestellt mit denen eine verteilte Anwendung vor Ausfällen von einzelnen Komponenten geschützt werden soll.

A. Redundanz

Redundanz ist eine einfach umzusetzende Möglichkeit ein System ausfallsicher zu machen und kann auf allen Ebenen, von den physikalischen Maschinen bis zur Anwendungslogik angewendet werden. Typisch sind mehrere Maschinen um sich gegen Hardwareausfälle zu schützen. In vielen Fällen wird die Anwendungslogik ebenfalls auf verschiedenen Maschinen deployt um im Falle eines Ausfalls von Maschinen nicht erst auf ein Deployment warten zu müssen. Darüber hinaus können dadurch Anfragen auf die laufenden Systeme verteilt werden, was die Last für das Gesamtsystem reduziert.

Hierzu wird ein Load Balancer benötigt. Für das Verwalten und Replizieren von Anwendungen auf mehreren Maschinen sind Tools wie Kubernetes oder Docker Swarm gemacht und verfügen dazu auch nativ über Load Balancer. Auch wenn Redundanz einfach umzusetzen ist, müssen Schwierigkeiten die mit dieser Lösung einhergehen beachtet werden. Wenn eine Anwendung mehrmals existiert kann es sein, dass ein Load Balancer Anfragen eines Nutzers auf verschiedene Instanzen der Anwendung verteilt. Bei Redundanz von Maschinen ist zu beachten, dass sich die Anwendungslogik nicht auf einen Zustand über den Benutzer im Hauptspeicher verlassen kann, da sich dieser mit Anfragen die von anderen Instanzen bearbeitet wurden, verändert haben kann. Zudem ist zu beachten, dass wenn ein Rechner ausfällt die Last auf die übrigen verlagert wird. Diese müssen über ausreichend Ressourcen verfügen um mit der neuen Last umgehen zu können. Sonst führt der Ausfall eines Rechners zum Ausfall aller anderen.

Um sich vor Verbindungsausfällen durch ISPs zu schützen können auch Internetverbindungen mittels Multihoming redundant gehalten werden. Hierbei ist ein Plattformanbieter mit mehreren ISPs verbunden und über verschiedene IP Adressen erreichbar. So kann der Plattformanbieter im Falle eines Ausfalls einer Verbindung zu einem bestimmten ISP über alternative Adressen erreicht werden [2].

Redundanz kann Ausfälle von Komponenten soweit transparent machen, dass andere Teile eines Systems diese nicht wahrnehmen. Dies ist allerdings nicht immer möglich. In diesem Fall müssen andere Anwendungen auf den Ausfall reagieren. Die folgenden Pattern sind Möglichkeiten, welche die Fehlerfortpflanzung in einem verteilten System verhindern sollen.

B. Retries, Timeout und Fallbacks

Microservice Anwendungen kommunizieren häufig über REST und damit über TCP Verbindungen [3]. Bei der Kommunikation können einige Fehler auftreten. So ist es beispielsweise möglich, dass eine bestehende Verbindung durch einen ausgefallenen Kommunikationspartner nicht wie in der Spezifikation beschrieben beendet wird. Dadurch hält der andere Partner ein Socket offen und blockiert eventuell einen Prozess der nichts ausführt dafür aber Hauptspeicherplatz und CPU Zeit verbraucht. Wenn mehrere solcher Prozesse existieren ist es möglich, dass die Ressourcen des Hosts so stark beansprucht werden, dass die Anwendung ebenfalls ausfällt oder Anfragen stark verzögert beantwortet werden.

Um dies zu verhindern ist es bei der Programmierung in verteilten Systemen von enormer Bedeutung einen Timeout zu setzen, also den Zeitraum in dem ein Kommunikationspartner Antworten muss, bevor ein Host die Verbindung abbricht um keine ungenutzten Ressourcen zu blockieren. Falsch konfigurierte Timeouts sind in Cloud Anwendungen eine häufige Fehlerursache, für schwerwiegende Ausfälle [4].

Die unkomplizierteste Möglichkeit auf den Fehler zu reagieren ist die fehlgeschlagene Anfrage ein weiteres Mal zu versenden. Problematisch ist dabei allerdings, dass ein stark belasteter Server durch Retries von Clients noch stärker belastet wird. Um dies zu verhindern können Fallbacks verwendet werden. Fallbacks sind alternativen zum ausgefallenen Service, die ein weniger präzises Ergebnis aber dafür überhaupt ein Ergebnis liefern. Typische Fallback Strategien sind der Zugriff auf einen Cache, oder das verwenden von zufälligen anstatt personalisierten Daten. In zeitkritischen Anwendungen kann ein definierter Timeout ebenfalls ein zu großer Zeitraum um den Ausfall eines Kommunikationspartners festzustellen. Um noch schneller auf Fehler zu reagieren, kann das Circuit Breaker Pattern verwendet werden.

C. Circuit Breaker

Das Circuit Breaker Pattern dient dem schnellen erkennen von Fehlern bei der Kommunikation mit anderen Hosts. Die Idee hinter dem Pattern entspricht dem Gedanken ”fail fast” [5] um möglichst schnell auf Anfragen antworten zu können. Dabei wird in Kauf genommen, dass von einem Fehler bei den Kommunikationspartnern ausgegangen wird, obwohl keine vorliegen. Dadurch verhindert das Pattern aber auch, dass ohnehin schon stark belastete Hosts durch weitere Anfragen noch weiter belastet werden.

Ein Circuit Breaker steuert und überwacht Aufrufe innherhalb eines Services zu anderen Services. Abhängig von der Häufigkeit mit der Anfragen zu langsam oder mit Fehlernachrichten beantwortet wurden, nimmt der Circuit Breaker einen anderen Zustand ein. Das Verhalten des Systems wird durch den aktuellen Zustand gesteuert. Dabei wird für jeden Service der angesprochen werden kann, ein eigener Zustandsautomat verwaltet. Die folgenden Zustände und Transitionen sind innerhalb eines Circuit Breakers möglich: *Geschlossen*: Der Circuit Breaker befindet sich initial im Zustand *Geschlossen* in diesem Zustand wird ein Zähler verwendet, der mit jeder fehlerhaft oder zu langsam beantworteten Anfrage an einen anderen Service hochgezählt wird. Dabei werden alle Anfragen an den jeweiligen Service gesendet. Wenn über einen definierten Intervall keine Fehler auftreten, kann der Zähler auch runtergezählt oder zurückgesetzt werden. Wenn der Zähler einen Schwellwert überschreitet

findet ein Zustandswechsel in den Zutsand *Offen* statt. *Offen*: Im Zustand offen werden keine Anfragen an den jeweiligen Service gesendet. Stattdessen wird der Aufrufer über einen Fehler informiert. Wie mit dieser Information verfahren wird ist nicht vorgeschrieben. Möglich sind Retries oder Fallbacks. Dieser Zustand gibt dem fehlerhaften System Zeit um den Fehler aufzulösen, ohne dass ihn weitere Anfragen erreichen. Zusätzlich werden Aufrufe schneller bearbeitet, da direkt über Fehler informiert werden kann. Der Circuit Breaker wechselt nach einer festgelegten Zeit in den Zustand *Halboffen*. *Halboffen*: Dieser Zustand dient der Überprüfung, ob das Fehlerhafte System wieder funktionstüchtig ist. Hierbei wird ein Teil der Aufrufer wie im Zustand Offen mit einer Fehlernachricht informiert, dass der Service nicht erreichbar ist. Für den anderen Teil der Aufrufer werden die Anfragen ausgeführt. Wenn diese ausreichend häufig korrekt bearbeitet werden, wird wieder in den Zustand *Geschlossen* gewechselt [5].

Circuit Breaker können auf verschiedene Arten eingesetzt werden. Jede Variante bietet eigene Vor- und Nachteile. Bibliotheken wie Hystrix oder Resilience4J bieten bereits fertige Implementierungen für Circuit Breaker auf der Client Seite. Ein Vorteil eines Clientseitigem Circuit Breakers ist, dass der Server bei einem Offenem Circuit Breaker keine Anfragen bearbeiten muss. Hierbei ist allerdings sicherzustellen dass auch jeder Client einen Circuit Breaker verwendet. Sonst ist der Schutz für Server nicht mehr gewährleistet.

Um Sicherzustellen, dass mit jeder Anfrage ein Circuit Breaker verwendet wird, kann ein serverseitiger Circuit Breaker eingesetzt werden. Hier ergibt sich allerdings der Nachteil, dass auch der Circuit Braker nicht mehr erreichbar ist, wenn der Server durch einen Hardwarefehler ausfällt, da beide Instanzen auf einer Maschine ausgeführt werden.

Ein Mittelweg aus client- und serverseitigem Circuitbreaker ist der Proxy Circuit Breaker. Hierbei sendet jeder Client eine Anfrage an einen Proxy, der den Circuit Breaker implementiert. Abhängig vom Zustand leitet dieser die Anfrage an den Server weiter, oder beantwortet diese sofort mit einer Fehlernachricht. Dies setzt voraus, dass alle Clients den Proxy kennen, und dass der Proxy die Antworten interpretieren kann [6].

D. Bulkheads

Beim Bulkhead Pattern werden Elemente einer Anwendung in Pools aufgeteilt, um Fehler zu isolieren. Auch dieses Muster lässt sich auf verschiedenen Ebenen umsetzen. Auch physische Redundanz kann als Umsetzung des Bulkhead Pattern betrachtet werden. Dabei können verschiedene Services auf unterschiedlichen Hosts ausgeführt werden. Hier wirkt sich ein Ausfall von Rechnern lediglich auf die Teile der Anwendung aus, die auf den ausgefallenen Instanzen ausgeführt wurden.

Auch auf Anwendungsebene kann das Bulkhead Muster umgesetzt werden. Typischerweise werden dafür verschiedene Threadpools gebildet, deren Threads für eine bestimmte Aufgabe zuständig sind. Ein standardmäßig konfigurierter Server verfügt über einen Threadpool aus dem Threads immer dann freigegeben werden, wenn diese angefordert wurden und verfügbar sind. Fehler die blockierte Threads erzeugen, führen wenn diese nicht rechtzeitig erkannt und behoben wurden dazu, dass alle Instanzen innerhalb eines Threadpools blockieren und die Anwendung keine Anfragen mehr bearbeiten kann. Mit dem Bulkhead Pattern wird für jede Aufgabe ein eigener Threadpool erzeugt. So blockieren lediglich die Threads die der fehlerhaften Funktionalität zugeordnet wurden. Die Requests die nicht die fehlerhafte Funktionalität anfragen, können weiter bearbeitet werden, da sie in anderen Threadpools laufen.

III. Chaos Engineering

Die zuvor beschriebenen Pattern sind einfache Möglichkeiten um Fehlerfortpflanzung innerhalb eines Systems zu verhindern. Allerdings setzen diese Methoden voraus, dass die Stellen an denen Fehler auftreten können bekannt sind. Wenn dies nicht der Fall ist können Fehler trotz umfangreichen Tests im Produktivsystem auftreten. Um diese Schwachstellen zu finden kann Chaos Engineering verwendet werden. Chaos Engineering wurde für die Plattform Netflix mit dem Ziel entwickelt das System robuster gegen Ausfälle einzelner Komponenten zu machen. Bei den Komponenten kann es sich um Hardware, das Netzwerk aber auch um Software handeln. Beim Chaos Engineering wird ein Chaos Experiment mit einem kleinen Anteil von Benutzern im laufenden System durchgeführt. Dabei werden Ausfälle von ausgewählten Komponenten simuliert um das Verhalten des Systems bei diesen Ausfällen zu beobachten. Da es sich um Experimente im laufenden System handelt wurden vier Prinzipien für Chaos Experimente definiert mit denen Schäden im Produktivsystem verhindert werden sollen. Darüber hinaus sollen die Prinzipien helfen einen möglichst hohen Informationsgewinn durch die Experimente zu erzielen [7].

A. Definition eines stabilen Zustands

Bevor beobachtet werden kann wie sich das System bei Ausfällen verhält, muss definiert sein wie das

System unter normalen Bedingungen funktioniert. Dazu müssen Metriken erhoben werden die den Zustand des Systems wiedergeben. Deshalb ist funktionierendes und gründliches Monitoring eine wichtige Grundvoraussetzung für Chaos Engineering. Bei der Auswahl der Metriken die das System beschreiben, ist zu beachten, dass sie die Erreichbarkeit des Systems darstellen. Deshalb sind Metriken wie CPU Auslastung oder Hauptspeicherverbrauch eher ungeeignet. Bei Ausfall von Rechnern ist zu erwarten, dass sich die umverteilte Last auf die Resourcen der noch funktionierenden Instanzen auswirkt. Von Interesse sind Metriken die zeigen ob der Benutzer, noch die Hauptfunktionalitäten einer Plattform nutzen kann. Deshalb wird der stabile Zustand bei Netflix durch die gestarteten Streams innerhalb von einer Sekunde definiert. Bei Verkaufsplattformen kann die Zahl der getätigten Käufe eine Metrik für den stabilen Zustand sein. Dennoch sollte nicht auf die Erhebung anderer Metriken verzichtet werden, da dadurch sichtbar wird, auf welche Komponenten sich ein Fehler auswirkt.

B. Auswahl von Fehlerfällen

Es gibt viele Möglichkeiten Fehler in einem Chaos Experiment zu simulieren, da Fehler auf allen Ebenen eines Systems, von der Hardware über den Hypervisor und den virtuellen Maschinen, bis zum Containerorchestrierungstool, auftreten können. Hierbei sollten Fehler nicht willkürlich gewählt, sondern priorisiert werden. Mögliche Indizien für die Bedeutsamkeit eines Fehlers sind die bisher aufgetretenen Fälle dieses Fehlers, die Eintrittswahrscheinlichkeit sowie der Schaden der dadurch entstehen kann. Dem gegenüber steht der Schaden der im Rahmen eines Chaos Experimentes entsteht. Um Ausfälle zu verursachen gibt es eine Reihe von Tools. Am prominentesten ist Chaos Monkey von Netflix, mit dem AWS Instanzen abgeschaltet werden können. Neben kompletten Ausfällen können auch Latenzen simuliert werden.

C. Ausführung der Fehler in Produktionsumgebung

Wenn ein stabiler Zustand und der zu simulierende Ausfall festgelegt wurde, kann ein Chaosexperiment durchgeführt werden. Dies findet innerhalb der Produktivumgebung statt. Begründet wird dies mit dem Gedanken, dass es Fehler geben kann die nur innerhalb dieser Umgebung stattfinden können. Es ist für Testfälle sehr schwer möglich das Produktivsystem direkt abzubilden, da sich Konfigurationen wie DNS Einträge von dem laufenden System unterscheiden. Zusätzlich werden Anfragen von echten Nutzern für die Chaos Experimente verwendet. Da eine Wechselwirkung zwischen fehlerhaftem System und dem Benutzerverhalten besteht. So führt eine Webseite mit hoher Ladezeit dazu, dass Benutzer häufiger den Refresh Button drücken, was die Last auf das getestete System weiter erhöht. Diese Effekte sind schwer vorhersagbar und damit auch schwer simulierbar, weshalb echte Nutzerdaten für die Tests verwendet werden.

D. Automatisierung von Experimenten

Unittests sind ein beliebtes Mittel in der klassischen Softwareentwicklung um sicherzustellen, dass Codeänderungen beispielsweise durch Refactoring oder Implementierung neuer Features, keine neuen Fehler hervorbringen. Neue Features oder Refactoring können allerdings auch neue Schwachpunkte in das System einbringen. Um gleichbleibende Robustheit bei neuen Deployments sicherzustellen, kann das Prinzip des kontinuierlichen Testens auch auf Chaos Experimente übertragen werden. Dazu kann mit jedem neuen Release eines Services, ein bereits konfiguriertes Chaos Experiment durchgeführt werden. Für die Automatisierung verwendet Netflix das Tool *Chaos Automation Platform (ChAP)* [8]. Hiermit ist es möglich vorkonfigurierte Experimente auszulösen. Hier zeigen sich auch einige Herausforderungen für die Zukunft der Automatisierten Chaos Experimente. ChAP übernimmt lediglich die Ausführung. Was noch automatisiert werden kann, ist die Konfiguration der Experimente sowie die Auswertung. Bei der Konfiguration besteht die Schwierigkeit darin Szenarien zu generieren, die auch einen Informationsgewinn schaffen. So lässt sich der Ursprung einer Fehlerkette nicht bestimmen wenn ein langsamer Prozessor in Kombination mit erhöhter Netzwerklatenz simuliert wird. Dennoch können beide Szenarien für sich verschiedene Symptome auslösen.

IV. Automatische Überwachung

Wie im vorherigen Kapitel erwähnt ist eine bestehende Überwachung eines Systems eine wichtige Grundlage für effektives Chaos Engineering. Doch nicht nur für Chaos Experimente, sondern auch für die frühzeitige Fehlererkennung ist dies essentiell. Für die Überwachung von Anwendungen existieren verschieden Ansätze, die sich gegenseitig nicht ausschließen. Monitoring ist das Sammeln von Metriken eines Systems um Fehler und das Eintreten eines Fehlers zu erkennen, um diese rechtzeitig beheben zu können. Beim Logging werden Informationen über den Status eines Systems und Ereignisse die in diesem stattfinden gesammelt. Typischerweise werden die Informationen für Menschen lesbar mit einem Zeitstempel dargestellt. Das Tracing dient dem Verfolgen des

Datenfluss innerhalb einer Anwendung. Traces sind vor allem bei der Fehlerlokalisation hilfreich, da diese die Fehler verursachende Codestelle aufdecken.

A. Monitoring

Beim Monitoring werden Metriken gesammelt um den Status eines Systems beobachten und Fehler aufdecken zu können. Monitoring lässt sich in zwei Arten unterteilen: Blackbox und Whitebox Monitoring [9].

Beim Whitebox Monitoring werden Daten direkt aus der Anwendung gesammelt. Um Whitebox Monitoring umsetzen zu können, muss auf den Quellcode der zu überwachenden Anwendung zugegriffen werden. Dabei ist das Exportieren der Metriken zwecks Monitoring eine Funktionalität der Anwendung. Hierbei lässt sich kritisieren, dass sich das Monitoring auf die Metriken auswirkt, da es ebenfalls innerhalb der Anwendung stattfindet. Ein Vorteil allerdings ist, dass sich Daten sammeln lassen die nicht durch Blackbox Monitoring erhoben werden können. Beispielsweise sind dies Zahl Datenbankanfragen in einer Minute, aber auch fachliche Informationen wie die Zahl der eingeloggten Nutzer.

Blackbox Monitoring ist das Überwachen des Systems in dem die Anwendung ausgeführt wird. Hierbei können keinerlei fachliche Informationen oder Implementierungsdetails überprüft werden. Der Gedanke hinter Blackbox Monitoring ist, dass sich Fehler in einer Anwendung auch in der Laufzeitumgebung bemerkbar machen. So lassen sich mit dem Überwachen von offenen Ports falsch konfigurierte Timeouts finden. Für Blackbox Monitoring muss also nicht der Anwendungscode verändert werden. Damit wird auch nicht das Verhalten der Anwendung beeinflusst.

Beide Verfahren schließen sich nicht aus und können zusammen verwendet werden. In beiden Fällen stellen sich allerdings besondere Herausforderungen in Microservice Umgebungen. Microservices werden in verteilten Systemen verwendet und sind, wenn Orchestrierungtools verwendet werden, nicht an bestimmte Maschinen für die Ausführung gebunden. Dadurch kann kein Blackbox Monitoring auf Betriebssystemebene stattfinden, da nicht sichergestellt ist, dass die Anwendung auch auf dem überwachtem System läuft. Zwar könnte dies erzwungen werden, verhindert aber die von Microservices gewünschte Flexibilität. Darüber hinaus werden häufig mehrere Instanzen einer Anwendung gleichzeitig ausgeführt. In solch einem Szenario reicht es nicht Metriken von einer Instanz zu sammeln um den Zustand des Systems bewerten zu können. Um eine umfassende Übersicht über den Zustand der Anwendung zu erhalten, müssen die Metriken aller Instanzen gesammelt und zusammengeführt werden.

Monitoring Systeme können nicht nur durch die Art der Daten die sie sammeln unterschieden werden, sondern auch durch das Verfahren mit dem die Daten vom Produzenten an dritte Systeme zur weiteren Verarbeitung weitergegeben werden. Hierbei wird zwischen den Push und Pull Verfahren unterschieden. Produzenten die das Push Verfahren verwenden, geben die Metriken bei Änderung der Werte selbstständig an dritte System weiter. Systeme die nach dem Pull Prinzip arbeiten, geben die Daten auf Anfrage an die auswertenden Komponenten. Wie in [10] beschrieben hat jedes Verfahren sowohl Vor- als auch Nachteile. Push basiertes Monitoring ermöglicht bei entsprechender Konfiguration eine konsistente Erfassung der Werte. Allerdings entsteht dadurch ein höheres Aufkommen an Netzwerkverkehr. Mit Pull basiertem Monitoring werden die Metriken typischerweise in definierten Zeitintervallen angefordert. Dadurch wird das Netzwerk weniger stark belastet. Allerdings ist es möglich das starke Schwankungen in den Metriken, die auf Fehler hinweisen können, verspätet oder gar nicht von den Datenkonsumenten erfasst werden.

B. Logging

Neben dem Monitoring also dem Sammeln von Metriken eines Systems mit dem Zweck dieses zu überwachen, ist Logging ein weiteres Mittel um Fehler in einem System frühzeitig erkennen zu können. Für Logging werden Informationen über das System in eine Datei geschrieben oder über die Standardausgabe ausgegeben. Der Unterschied zum Monitoring besteht darin, dass die Informationen in für Menschen lesbaren Texten dargestellt werden. Hier können neben Metriken, Ereignisse wie Fehler, Methodenaufrufe oder das Starten von Hintergrundprozessen mitgeteilt werden. Neben der Fehlererkennung können Logs auch für die Analyse von Nutzerverhalten verwendet werden. Wie auch beim Monitoring besteht auch hier die Herausforderung, dass die Logs von allen Instanzen eines Services gesammelt und zusammengefasst werden müssen, um eine Übersicht über das Gesamtsystem zu erhalten. Bei Logging ist zu beachten, dass die Log Nachrichten in einem einheitlichen Format ausgegeben werden, damit diese automatisiert analysierbar sind. Dass große Datenmengen während des Logging entstehen ist bei Plattformen mit viel Nutzeraktivität nicht unüblich. Deshalb werden wie in [11] beschrieben Architekturen für die Auswertung von Logs entwickelt. Häufig werden dabei Streamingplattformen eingesetzt.

C. Tracing

Tracing wird verwendet um den Programmfluss eines Systems nachvollziehbar zu machen. Ein typisches Einsatzszenario für Tracing ist das Debugging. Hierbei wird der Verlauf einer Anwendung untersucht, um die Programmstellen zu finden die Fehler verursachen. In monolithischen Anwendungen reicht ein Stacktrace um den gesamten Verlauf einer Anwendung untersuchen zu können. In einer Microservice Umgebung ist dies nicht ohne weitere Tools möglich, da Komponenten über das Netzwerk kommunizieren. Für das verteilte Tracing existieren einige Tools, wie Jaeger oder Istio. Wie bei Monitoring kann auch bei Tracing eine Unterscheidung in Black und Whitebox gemacht werden.

Bei Blackbox Tracing wird lediglich der Paketfluss zwischen zwei Komponenten beobachtet, um zu erfahren welche Komponenten mit welcher Häufigkeit kommunizieren. Hierbei lässt sich kein Zusammenhang zwischen einzelnen Paketströmen herstellen.

Für Whitebox Tracing werden Pakete bevor sie über das Netzwerk verschickt werden in der Anwendung oder durch eine Bibliothek die als Middleware fungiert mit Informationen, wie einer eindeutigen ID für jede Anfrage, angereichert. Auf der Empfängerseite werden die Zusatzinformationen an eine zentrale Stelle geleitet die aus der Gesamtheit aller Daten Traces für das verteilte System bildet. So kann der Kontrollfluss für eine Anfrage in einer verteilten Anwendung über mehrere Knoten beobachtet werden. Darüber hinaus ist hierbei der Kontext der Pakete, also die Daten die verarbeitet werden bekannt. Dadurch lässt sich bestimmen unter welchen Bedingungen welche Services miteinander kommunizieren [12].

V. Alerting

Daten die aus dem Monitoring gewonnen wurden können von Systemen für Alerting verwendet werden. Beim Alerting überwacht ein System mit Hilfe der gesammelten Metriken eine laufende Anwendung und alarmiert Verantwortliche falls Fehler auftreten, oder falls die Gefahr besteht, dass Fehler auftreten werden. Typischerweise wird über Regeln definiert, wann das Alerting System einen Alarm auslösen soll. Diese Regeln beziehen sich häufig auf Schwellwerte. Eine im Augenblick erforschte Alternative zu Regelbasierten Alerting Systemen sind Modelbasierte Alerting Systeme. Ziel ist es durch die Modelle ein effizienteres Alerting, mit weniger Fehlalarmen und besserer Fehlererkennung, als regelbasierte Systeme bieten zu ermöglichen. Hierbei werden Machine Learning Methoden eingesetzt, in [13] beispielsweise Support Vector Machines, um ein Model zu trainieren, dass über Fehler informiert oder diese sogar voraussagt. Wenn Fehler vorhergesagt werden können, ist es möglich Gegenmaßnahmen einzuleiten.

VI. Zusammenfassung und Ausblick

In dieser Arbeit wurden Pattern vorgestellt, mit denen Systeme widerstandsfähiger und zuverlässiger gemacht werden können. Darüber hinaus wurde mit Chaos Engineering als Ergänzung zu klassischen Tests gezeigt wie Schwachstellen in einer verteilten Anwendung identifiziert werden können. Zusätzlich wurden in den letzten beiden Kapiteln Möglichkeiten für die Überwachung laufender Systeme vorgestellt.

Hier wurde deutlich, dass Themen wie die Automatisierung von Chaos Experimenten oder Alerting Raum für Untersuchen geben.

Für die Folgenden Semester soll eine Beispielanwendung mit funktionierendem Monitoring umgesetzt werden, mit der die vorgestellten Techniken angewendet und evaluiert werden sollen. Denkbar ist es für das Grundprojekt die vorgestellten Pattern bezüglich der Parametrisierung mit Hilfe von Chaos Experimenten zu untersuchen. Da diese Versuche, wenn gründliches Monitoring umgesetzt wurde, große Datenmengen erzeugen, soll im Hauptprojekt Automatisierung, also die Generierung und Auswertung, von Chaos Experimenten untersucht werden.

References

[1] J. Petrić, T. Hall, and D. Bowes, "How effectively is defective code actually tested?: An analysis of junit tests in seven open source systems," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE'18. New York, NY, USA: ACM, 2018, pp. 42–51.

[2] J. Abley, B. Black, and V. Gill, "Goals for ipv6 site-multihoming architectures," Internet Requests for Comments, RFC Editor, RFC 3582, August 2003.

[3] A. Archip, C. Amarandei, P. Herghelegiu, C. Mironeanu, and E. ?erban, "Restful web services ? a question of standards," in *2018 22nd International Conference on System Theory, Control and Computing (ICSTCC)*, Oct 2018, pp. 677–682.

[4] T. Dai, J. He, X. Gu, and S. Lu, "Understanding real-world timeout problems in cloud server systems," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, April 2018, pp. 1–11.

[5] M. T. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2018.

[6] F. Montesi and J. Weber, "From the decorator pattern to circuit breakers in microservices," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing - SAC '18*. Pau, France: ACM Press, 2018, pp. 1733–1735. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3167132.3167427

[7] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos Engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, May 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7436642/

[8] A. Basiri, A. Blohowiak, L. Hochstein, and C. Rosenthal, "A platform for automating chaos experiments," *CoRR*, vol. abs/1702.05849, 2017. [Online]. Available: http://arxiv.org/abs/1702.05849

[9] C. Ciccotelli, L. Aniello, F. Lombardi, L. Montanari, L. Querzoni, and R. Baldoni, "Nirvana: A non-intrusive black-box monitoring framework for rack-level fault detection," in *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*, Nov 2015, pp. 11–20.

[10] H. Huang and L. Wang, "P & P: A Combined Push-Pull Model for Resource Monitoring in Cloud Computing Environment," in *2010 IEEE 3rd International Conference on Cloud Computing*, Jul. 2010, pp. 260–267.

[11] G. M. Evgenyevich, B. A. Valerievich, and B. M. Alekseevna, "Using apache spark to collect analytic from the streaming data processing application logs," in *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, June 2018, pp. 1–4.

[12] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," p. 14.

[13] A. Bhattacharyya, H. Singh, S. A. J. Jandeghi, and C. Amza, "Online detection of anomalous applications on the cloud," in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '17. Riverton, NJ, USA: IBM Corp., 2017, pp. 161–169. [Online]. Available: http://dl.acm.org/citation.cfm?id=3172795.3172814