Fehlertoleranzanalyse von Microservice basierten Softwarearchitekturen

Konzept und Anwendung am JValue ODS

MASTERARBEIT

Jonas Schüll

Eingereicht am 11. Mai 2021

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Betreuer:
Prof. Dr. Dirk Riehle, M.B.A.
Georg Schwarz, M.Sc.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 11. Mai 2021

License

Erlangen, 11. Mai 2021

Abstract

Microservice-based software architectures play an essential role in building sizeable scalable cloud systems. The main advantage of microservices compared to the traditional software monoliths is the independent development, deployment, and scaling of the individual microservices, which allows innovations at a higher speed. Because microservice-based architectures are distributed systems, complexity is shifted from code to the network and communication layer. Therefore, additional failures like service outage or network connectivity loss arise, which must be tolerated to keep the system healthy and running.

Within this thesis, a reusable concept is developed to analyse the fault tolerance of microservice-based software architectures. This allows for revealing weaknesses in the architecture that negatively affects the system's reliability and resilience. For frequent problems, solution proposals are provided. The concept's applicability and effectiveness are evaluated by applying it at the JValue Open Data Service (ODS). The analysis revealed several issues regarding the ODS's fault tolerance, which could be fixed with the provided solutions.

Kurzfassung

Für stark skalierbare Cloud basierte Softwaresysteme stellen Microservices einen vielversprechenden Architekturansatz dar. Der wesentliche Vorteil von Microservices gegenüber den klassischen Softwaremonolithen ist die Unabhängigkeit der Entwicklung, des Deployments und der Skalierung. Durch diese Unabhängigkeit ist eine höhere Innovationsgeschwindigkeit möglich. Allerdings handelt es sich bei Microservice basierten Architekturen um verteilte Systeme. Aufgrund dessen ist die Komplexität von der Codeebene in die Netzwerk- und Kommunikationsebene verschoben. Hieraus ergeben sich neue Fehlerquellen, wie zum Beispiel der Ausfall einzelner Dienste oder der Netzwerkverbindung zwischen den Microservices. Diese Fehlerquellen müssen entsprechend toleriert werden, damit das gesamte System betriebsfähig bleibt und die Unabhängigkeit der Microservices auch im Sinne der Fehlertoleranz gewährleistet ist.

In dieser Arbeit wird ein wiederverwendbares Konzept entwickelt, um Microservice basierte Softwarearchitekturen hinsichtlich ihrer Fehlertoleranz zu analysieren. Mit Hilfe dieses Konzeptes können Schwachstellen in der Architektur aufgedeckt werden, die die Ausfallsicherheit und Robustheit des Systems negativ beeinflussen. Zusätzlich werden entsprechende Lösungsmöglichkeiten vorgestellt, um die gefundenen Probleme zu beheben. Die Anwendbarkeit und Wirksamkeit des Konzeptes wird am Beispiel des JValue Open-Data-Service (ODS) evaluiert. Die Analysen konnten verschiedene Probleme in Bezug auf die Fehlertoleranz des ODS identifizieren, die mit Hilfe der bereitgestellten Maßnahmen beseitigt wurden.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AMQP	Advanced-Message-Queuing-Protocol
CDC	Change-Data-Capture
DNS	Domain-Name-System
ETL	Extract, Transform, Load
FAU	Friedrich-Alexander-Universität Erlangen-Nürnberg
HTTP	Hypertext-Transfer-Protocol
ODS	Open-Data-Service
REST	Representational-State-Transfer
SRV-Record	Service-Record

Kapitel 1 Einleitung

Einer aktuellen Umfrage des O'Reilly Verlags bezüglich der Verwendung von Microservices setzen 61% der befragten Firmen seit ein bis 5 Jahren Microservices ein (Loukides & Swoyer, 2020). Dieses Interesse ist vor allem auf die Erfolgsgeschichten von Amazon (O'Hanlon, 2006) und Netflix (Evans, 2016) zurückzuführen, die schon seit über 10 Jahren Microservices sehr erfolgreich einsetzen und sehr viel über ihre positiven Erfahrungen berichten. Zu den Vorteilen von Microservices zählen starke Modularisierung, größere Technologiefreiheit und unabhängige Skalierbarkeit sowohl zur Laufzeit als auch während der Entwicklung in mehreren Teams (Wolff, 2018). Doch es gibt auch Nachteile dieses Architekturstils, die vor allem auf die Komplexität von verteilten Systemen zurückzuführen sind. Die Kommunikation zwischen Microservices erfolgt über Netzwerke, die langsamer und unzuverlässiger sind als die Kommunikation zwischen Prozessen oder innerhalb eines Prozesses. Des Weiteren ist es aufgrund der Verteilung sehr schwer starke Konsistenz zu garantieren und Ausfälle von Diensten müssen toleriert werden, um eine Ausbreitung dieser Ausfälle zu verhindern (Fowler, 2015). Diese Komplexität ist vielen Softwareentwicklern und -architekten oft nicht bewusst, da sie häufig wenig Erfahrung mit verteilten Systemen haben.

In dieser Arbeit soll ein Konzept entwickelt werden, mit dem Probleme hinsichtlich der Fehlertoleranz und Ausfallsicherheit in Microservice basierten Architekten erkannt werden können. Dabei wird eine Architektur nach einem strukturierten Vorgehen analysiert, welches Fehlerquellen von verteilten Systemen aufspürt. Außerdem werden Lösungsvorschläge bereitgestellt, um häufige Fehler ohne tiefgehendes Verständnis beheben zu können. Das Konzept verwendet einen Abhängigkeitsgraphen, um problematische Stellen in der Architektur zu lokalisieren. Es werden verschiedene Kommunikationsmuster unterstützt, sodass auch Event getriebene Architekturen von der Methodik profitieren können. Das Konzept wird abschließend am JValue Open-Data-Service (ODS) evaluiert. Der ODS verwendet eine Microservices Architektur mit Event basierter Kommunikation zwischen den einzelnen Diensten. Funktional ist der ODS eine Plattform, mit der Open-Data-Quellen automatisiert abgefragt werden können. Zusätzlich können diese Daten mit Hilfe von Pipelines in andere Formate transformiert oder einzelne Daten extrahiert werden.

Kapitel 2 Anforderungen

Da in dieser Arbeit ein Konzept zur Fehlertoleranzanalyse erstellt wird, müssen zuerst die Anforderungen für dieses Konzept und dessen Anwendung am JValue Open-Data-Service (ODS) definiert werden. Die Anforderungen wurden in Gesprächen mit den Entwicklern des ODS gesammelt und durch eigene Erfahrungen mit dem ODS erweitert. Bei dieser Arbeit handelt es sich um eine explorative Arbeit, bei der die exakten Analyseverfahren erst noch erarbeitet werden müssen. Daher können einige der folgenden Anforderungen nicht spezifischer definiert werden. Es wird zuerst auf die allgemeinen, dann auf die funktionalen und als letztes auf die nichtfunktionalen Anforderungen eingegangen.

2.1 Allgemeine Anforderungen

Das allgemeine Ziel dieser Arbeit ist es, ein Konzept zu erarbeiten, um in Microservice basierten Softwarearchitekturen Probleme in Bezug auf Ausfallsicherheit und Fehlertoleranz zu identifizieren und zu beheben. Die Anwendbarkeit und Wirksamkeit dieses Konzeptes soll am ODS evaluiert werden. Außerdem sollen entsprechende Lösungen implementiert werden, um den ODS ausfallsicherer und fehlertoleranter zu machen.

2.2 Funktionale Anforderungen

Im folgenden Kapitel werden die funktionalen Anforderungen sowohl für das Konzept als auch für die Anwendung am ODS dargestellt.

2.2.1 Funktionale Anforderungen an das Konzept

Als Grundlage für die Fehlertoleranzanalyse dient ein Graph, der die Abhängigkeitsbeziehung zwischen den Microservices modelliert. Hierbei soll die verwendete

Kommunikationsart zwischen zwei Microservices die Abhängigkeit definieren. Der Abhängigkeitsgraph muss nicht nur die Anfrage-Antwort basierte Kommunikation unterstützen, sondern auch die bei Microservices oft eingesetzte Nachrichten basierte Kommunikation. Das Konzept soll Muster bereitstellen, mit denen in diesem Graph Probleme in Bezug auf die Ausfallsicherheit und Fehlertoleranz identifiziert und lokalisiert werden können. Außerdem sollten passende Lösungsvorschläge dargelegt werden, mit denen die gefundenen Probleme behoben werden können.

Zusätzlich war es gewünscht die Methoden des Chaos-Engineering anzuwenden. So können zum einen die Mechanismen zur Ausfallsicherheit und Fehlertoleranz von Microservices überprüft und zum anderen weitere Probleme entdeckt werden.

2.2.2 Funktionale Anforderungen für die Erprobung des Konzeptes am ODS

Um das Konzept zu evaluieren, soll dieses am JValue ODS erprobt werden. Ziel ist es mit Hilfe der Analysen aus dem Konzept Probleme in Bezug auf die Ausfallsicherheit und Fehlertoleranz zu identifizieren. Für die gefundenen Probleme werden entsprechende Lösungen implementiert, wobei jeweils die Konsistenzgarantien gegen die Verfügbarkeitsgarantien abgewogen werden sollen. Das Ziel ist es, den ODS fehlertoleranter und ausfallsicherer zu machen. Vor allem dürfen sich Ausfälle und Fehler nicht auf andere Dienste ausbreiten und wiederkehrende Fehler müssen so behandelt werden, dass diese das System nicht unnötig blockieren. Danach wird die Wirksamkeit der implementierten Lösungen mit Hilfe von Chaos-Engineering-Experimenten überprüft.

2.3 Nichtfunktionale Anforderungen

Nachdem die funktionalen Anforderungen dargelegt worden sind, werden im Folgenden die nichtfunktionalen Anforderungen beschrieben.

2.3.1 Nichtfunktionale Anforderungen an das Konzept

Das Konzept muss wiederverwendbar sein, damit es auf andere Microservice basierte Architekturen übertragbar ist. Da das Konzept in dieser Arbeit nur am ODS erprobt wird, sollen nicht nur ODS spezifische Fehler, sondern auch verallgemeinerbare Fehler gefunden werden.

Um den Einstieg in das komplexe Thema verteilte Systeme und Fehlertoleranz zu erleichtern, muss das Konzept einfach anzuwenden sein. Dafür wird das Konzept in kleine Schritte unterteilt, die nacheinander angewendet werden können.

2.3.2 Nichtfunktionale Anforderungen an die Implementierung von Lösungen am ODS

Bei der Umsetzung des Konzeptes am ODS muss darauf geachtet werden, dass die gesamte Funktionalität des ODS erhalten bleibt. Somit kann gewährleistet werden, dass Benutzer des ODS keine Einschränkungen oder Unterbrechungen während der Implementierung der Lösungen erfahren.

Zusätzlich soll die Entwicklungskomplexität niedrig gehalten werden. Die Auswahl bei alternativen Lösungen sollte mehr fachlich als technisch motiviert sein, da in den meisten Fällen fachliche Lösungen weniger komplex sind.

Die verschiedenen Lösungsmöglichkeiten für die gefunden Probleme sollen den anderen Entwicklern präsentiert und mit ihnen diskutiert werden, um eine passende Lösung auszuwählen. Des Weiteren müssen alle Änderungen am ODS abgesprochen werden, damit andere Beteiligte nicht unnötig eingeschränkt werden.

Kapitel 3 Grundlagen

In diesem Kapitel werden die Theorie und die Grundlagen für die Entwicklung des Konzeptes zur Fehlertoleranzanalyse dargestellt. Dabei wird zuerst auf Microservices, Abhängigkeitsgraphen und verteilte Systeme eingegangen. Als Letztes werden die Prinzipien des Chaos-Engineerings betrachtet.

3.1 Microservices

Die großen Internetfirmen, wie zum Beispiel Amazon und Netflix, sind mit monolithischen Anwendungen schnell an die Grenzen der Skalierbarkeit gestoßen. Jamshidi, Pahl, Mendonca, Lewis und Tilkov (2018) beschreiben dabei nicht nur Probleme auf technischer Ebene, sondern auch personell war die Koordination der Entwicklungsteams schwierig, da alle Beteiligten gleichzeitig an einer Anwendung arbeiten mussten. Daher unterteilen diese Firmen ihre monolithischen Anwendungen in viele kleine Dienste, die jeweils von einem Team entwickelt werden und nur für eine bestimmte Aufgabe zuständig sind.

Fowler und Lewis (2014) definieren eine Softwarearchitektur mit vielen kleinen Diensten als Microservices. Jeder Microservice läuft in seinem eigenen Prozess und kommuniziert mit den anderen Diensten über eine Netzwerkverbindung. Außerdem ist es für Microservices charakteristisch, dass sie unabhängig von den anderen Diensten in Produktion gebracht werden können. Dieser Prozess erfolgt oft vollständig automatisiert. Wolff (2018) beschreibt in seinem Buch „Microservices - Grundlagen flexibler Softwarearchitekturen“ die Vor- und Nachteile von Microservice basierten Softwarearchitekturen, die im Folgenden aufgezeigt werden.

Vorteile

- Das gesamte System wird in unabhängige Dienste aufgeteilt.
- Einzelne Teile der gesamten Anwendung können leicht ersetzt werden.

- Die Microservices können unabhängig entwickelt, skaliert und in Produktion gebracht werden.
- Die Microservices sind technologisch unabhängig, sodass zum Beispiel unterschiedliche Programmiersprachen oder Frameworks verwendet werden können.

Nachteile

- Die Kommunikation über das Netzwerk ist nicht zuverlässig.
- Die Kommunikation über das Netzwerk besitzt eine Latenz.
- Ein verteiltes System weist eine höhere technische Komplexität auf, da es mit Teilausfällen umgehen muss. So kann der Absturz eines einzelnen Microservices Fehler bei anderen Diensten hervorrufen oder deren Funktionalität einschränken, sodass unterschiedliche Teile des gesamten Systems weiterhin funktionieren und andere nicht mehr.
- Eine komplett isolierte Entwicklung der einzelnen Microservices kann es nicht geben, da jegliche Kommunikation zwischen zwei Diensten eine Abhängigkeit zwischen diesen verursacht.
- Die technologische Unabhängigkeit ist auch ein Nachteil, da so sehr viele unterschiedliche Technologien in einem System eingesetzt werden können.

3.1.1 Service-Discovery

Microservices müssen die IP-Adresse und die Portnummer von anderen Microservices herausfinden können, um mit diesen zu kommunizieren. Am Einfachsten lässt sich die Service-Discovery mit statischen Schlüssel-Wert-Paaren realisieren. Diese enthalten für jeden Microservice die dazugehörige IP-Adresse sowie die Portnummer und werden entweder beim Bauen oder beim Starten im aufrufenden Microservice hinterlegt. Dieses Verfahren hat allerdings einen großen Nachteil, wenn sich Änderungen ergeben, wie zum Beispiel eine Änderung der IP-Adresse oder das Hinzufügen eines neuen Microservices. In diesen Fällen muss die Information auf alle bestehenden Microservices angepasst werden und die Microservices können nicht mehr unabhängig in Produktion gebracht werden. (Wolff, 2018)

Als weitere Möglichkeit kann das Domain-Name-System (DNS) für die Service-Discovery verwendet werden, welches im Internet dafür zuständig ist, einen Hostname, wie `www.fau.de`, in die IP-Adresse des dazugehörigen Servers aufzulösen.

Auch in einer Microservices Umgebung kann DNS dafür genutzt werden, die IP-Adresse zu einem Microservice zu liefern. Jedem Microservice wird dann ein eindeutiger Hostname zugewiesen, der von anderen Microservices verwendet werden kann, um mittels DNS die entsprechende IP-Adresse abzufragen. Da DNS standardmäßig im A-Record nur die IP-Adresse zurückgibt, muss ein entsprechender Service-Record (SRV-Record) mit der Portnummer angelegt werden. Somit können Microservices die IP-Adresse und den Port von anderen Diensten über DNS abfragen. DNS bietet zudem Unterstützung für Caching, sodass die Auflösung beschleunigt werden kann. Außerdem ist DNS ausfallsicher, da es mehrere DNS-Server geben kann, die einen Namensraum verwalten. (Wolff, 2018)

Des Weiteren kann ein eigenständiger Service-Discovery-Dienst verwendet werden. Bei solchen Diensten kann ebenfalls die IP-Adresse und der Port für einen Microservice angefragt werden. Oft unterstützen sie weitere Funktionen wie zum Beispiel eine integrierte Lastverteilung (siehe Abschnitt 3.1.3 Lastverteilung). Außerdem kann ein Mechanismus zur Erkennung, ob eine Instanz eines Dienstes verfügbar ist, eingebaut sein. In diesem Fall kann entweder eine andere Instanz ausgewählt werden oder ein Fehler zurückgegeben werden, sodass keine unnötigen Anfragen an den Dienst geschickt werden. Zudem bieten einige Service-Discovery-Dienste an, weitere Metadaten und Konfigurationen zu den Microservices zu hinterlegen, die ebenfalls abgefragt werden können. Solche eigenständige Service-Discovery-Dienste sind zum Beispiel Netflix Eureka[1] oder Consul[2]. (Wolff, 2018)

3.1.2 Skalierung

Da die Skalierung von Diensten bei Microservices von zentraler Bedeutung ist, wird dies im folgenden Abschnitt beschrieben. Skalierung ist notwendig, wenn die zur Verfügung stehenden Ressourcen des Servers für den Dienst nicht mehr ausreichen. Zum Beispiel kann die CPU-Geschwindigkeit zu gering sein oder der Arbeitsspeicher reicht nicht mehr aus, sodass die korrekte Funktionsweise des Dienstes nicht mehr gewährleistet ist. In diesen Fällen muss der Dienst skaliert werden. Dienste können vertikal oder horizontal skaliert werden. Der Unterschied ist in der Abbildung 3.1 dargestellt. Bei vertikaler Skalierung wird der Dienst auf einem Server mit mehr Ressourcen gestartet. Dies ist prinzipiell die einfachere Skalierungsart. Allerding ist zu bedenken, dass vertikales Skalieren eine Grenze hat, da Server nicht mit unendlich vielen Ressourcen ausgestattet werden können.

Bei horizontaler Skalierung hingegen werden mehrere Instanzen des Dienstes gestartet. Somit kann die Last auf mehrere Instanzen aufgeteilt werden. Dies funk-

[1] https://github.com/Netflix/eureka
[2] https://www.consul.io

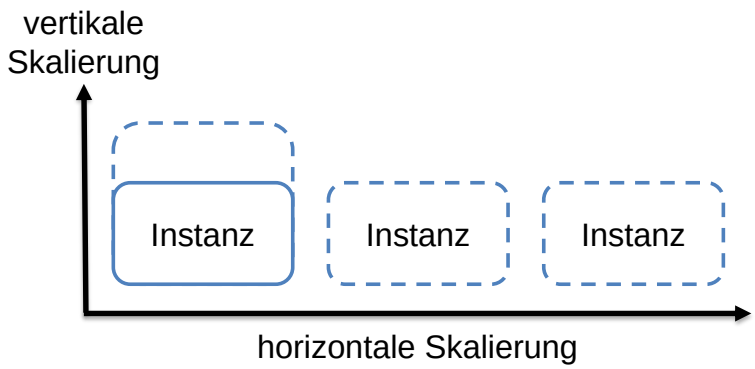


Abbildung 3.1: Unterschied zwischen horizontaler und vertikaler Skalierung
Quelle: in Anlehnung an Wolff (2018)

tioniert in der Regel nur für Dienste, die keinen Zustand haben, da es hier egal ist, welche Instanz eine Anfrage bearbeitet. Dienste mit einem Zustand, wie zum Beispiel Datenbanken, können nicht ohne weiteres horizontal skaliert werden, da sonst jede Instanz nur ihren eigenen Zustand verwalten würde. Es wird zusätzlich ein Mechanismus benötigt, der die Zustandsänderungen zwischen den einzelnen Instanzen überträgt, sodass alle Instanzen immer den gleichen Zustand haben. Dies wird als Replikation bezeichnet und verschiedene Replikationsmöglichkeiten werden im Abschnitt 3.3.3 dargestellt. Zusätzlich hat die horizontale Skalierung den Vorteil, dass Ausfälle von Instanzen durch andere Instanzen leicht kompensiert werden können. Hierfür wird nur ein Mechanismus benötigt der Ausfälle erkennt und anschließend alle eingehenden Anfragen an andere funktionierende Instanzen umleitet. Das Erkennen von Ausfällen kann zum Beispiel mit Hilfe eines Circuit-Breaker (siehe Abschnitt 3.1.4) und das Umleiten von Anfragen mit Hilfe einer Lastverteilung (siehe Abschnitt 3.1.3) erfolgen. Ein weiterer Vorteil von horizontaler Skalierung ist, dass die Anzahl der Instanzen dynamisch angepasst werden kann, je nachdem wie groß die Last ist. Wenn es zum Beispiel eine Lastspitze gibt, können weitere Instanzen gestartet werden, um die bestehenden Instanzen nicht zu überlasten. Sobald die Last wieder abgenommen hat, können überflüssige Instanzen wieder gestoppt werden, um Kosten zu sparen. Solche Mechanismen und die Aufteilung der eingehenden Anfragen auf die einzelnen Instanzen ist Aufgabe der Lastverteilung, die im folgenden Abschnitt erläutert wird. (Newman, 2020; Wolff, 2018)

3.1.3 Lastverteilung

Bei der Lastverteilung geht es darum die Anfragen möglichst effizient auf die vorhandenen Instanzen aufzuteilen. Da die Lastverteilung genau über die Menge der eingehenden Anfragen informiert ist, ist sie oft auch dafür verantwortlich die

horizontale Skalierung dynamisch je nach Last zu steuern. Wolff (2018) beschreibt drei Möglichkeiten die Lastverteilung zu implementieren.

Proxy basierte Lastverteilung Hierfür ist ein separater Proxy-Server notwendig, durch den alle eingehenden Anfragen laufen. Dieser leitet jeweils die Anfragen nach dem eingestellten Verteilungsverfahren an die entsprechenden Instanzen weiter. Es kann entweder einen globalen Proxy-Server geben oder einen Proxy-Server pro Microservicetyp.

In der Service-Discovery Ein Service-Discovery-Dienst kann auch eine Lastverteilung integriert haben. Der Service-Discovery-Dienst wählt dann bei einer Anfrage eine konkrete Instanz des Microservice aus und liefert dessen IP-Adresse und die Portnummer aus.

Client seitige Lastverteilung Bei der dritten Art von Lastverteilung wird kein weiterer Dienst benötigt, sondern die Lastverteilung und Auswahl einer Instanz findet auf der Clientseite statt. Hierzu benötigt der Client vorab eine Liste mit allen Instanzen. Deswegen eignet sich dieses Verfahren nicht bei dynamischer Skalierung.

3.1.4 Resilienz

In einer Microservice basierten Architektur sollte ein Ausfall oder eine Störung eines Dienstes möglichst wenig Auswirkung auf die anderen Microservices haben. Ist dies nicht der Fall, kann der Ausfall eines einzelnen Dienstes weitere Ausfälle verursachen und im schlimmsten Fall das gesamte System lahmlegen. „Aus diesem Grund muss ein Microservice gegen den Ausfall anderer Microservices abgesichert werden. Diese Eigenschaft nennt man Resilienz.“ (Wolff, 2018). Dafür beschreibt Wolff (2018) in seinem Buch die folgenden Maßnahmen:

Timeout Timeouts können verwendet werden, um ausgefallene oder überlastete Dienste zu erkennen. Hierbei werden Anfragen mit einem Fehler abgebrochen, wenn nach einer bestimmten Zeit keine Antwort zurückgeliefert wurde.

Circuit-Breaker Ein Circuit-Breaker funktioniert wie eine Sicherung in einem Stromkreis. Wenn ein Dienst ausgefallen ist oder Fehler zurückliefert, verhindert der Circuit-Breaker, dass weitere Anfragen an diesen Dienst gesendet werden. Hierfür hat er zwei Zustände. Im geschlossenen Zustand lässt er alle Anfragen an

den Microservice durch. Wenn Fehler eintreten, wird abhängig von einer eingestellten Fehlerfrequenz der Circuit-Breaker geöffnet und er blockiert alle folgenden Anfragen mit einem Fehler. Das entlastet den ausgefallenen Dienst, sodass er Zeit hat, sich zu erholen oder neugestartet werden kann. Nach einer konfigurierbaren Zeit schließt sich der Circuit-Breaker und er leitet wieder alle Anfragen an den Microservice weiter.

Bulkhead Der Begriff „Schott“ (auf Englisch „Bulkhead“) stammt aus dem Schiffsbau und bezeichnet eine Tür, die wasserdicht verschlossen werden kann. Schotte unterteilen das Schiff in mehrere Teile, sodass bei Wassereintritt nur ein Teil des Schiffes betroffen ist und eine Havarie verhindert werden kann. In Microservice basierten Architekturen kann diese Technik auch eingesetzt werden, um Fehler auf bestimmte Bereiche zu beschränken. Zum Beispiel können bestimmte Clients immer mit der gleichen Instanz eines Microservice verbunden werden, sodass ein Ausfall einer Instanz nur einen Teil der Clients betrifft.

Steady-State Steady-State bezeichnet die Eigenschaft, dass ein System unendlich laufen kann. Dies bedeutet zum Beispiel, dass das System ein korrektes Speichermanagement betreibt und auch nicht mehr benötigte Dateien zeitnah löscht. So kann sichergestellt werden, dass nicht irgendwann der gesamte Hauptspeicher oder Plattenspeicher ausgenutzt ist. Häufig werden beispielsweise Log-Dateien nur geschrieben und alte nicht mehr benötigte Log-Dateien werden nicht gelöscht. Auch sollten Einträge in Caches gelöscht werden, um Platz für neue Einträge zu schaffen und um die Speicherkapazität nicht zu erschöpfen.

Fail-Fast Im Fehlerfall sollten Anfragen immer so schnell wie möglich mit einer Fehlermeldung beantwortet werden. Dies hat zum einen den Vorteil, dass weniger Ressourcen verschwendet werden, zum anderen kann früher und schneller auf einen Fehler reagiert werden.

Handshaking Als Handshaking wird der Prozess bezeichnet, der eine Kommunikation einleitet. Hierbei kann ein Microservice auch eine Anfrage ablehnen, wenn es zum Beispiel aktuell nicht genügend freie Ressourcen gibt, um diese Anfrage zu bearbeiten. So können Timeouts vermieden werden, wenn ein Microservice eine hohe Last an Anfragen hat.

3.1.5 Kommunikation

In diesem Abschnitt werden verschiedene Kommunikationsarten in Microservice basierten Architekturen beschrieben. Diese erfolgen immer über ein Netzwerk, das alle Microservices verbindet. Grundsätzlich kann zwischen Anfrage-Antwort basierter und Nachrichten basierter Kommunikation unterschieden werden. Im Folgenden werden diese genauer dargestellt und jeweils ihre Vor- und Nachteile erläutert.

Anfrage-Antwort basierte Kommunikation

Bei der Anfrage-Antwort basierten Kommunikation sind immer genau zwei Dienste beteiligt. Der eine Dienst stellt dabei eine Funktionalität bereit, die der andere Dienst mit ein Anfrage aufrufen kann. Nach der Bearbeitung der Anfrage sendet der angefragte Dienst eine Antwort zurück an den anfragenden Dienst. Diese Antwort kann auch keine Daten enthalten und signalisiert dann nur, dass die Antwort erfolgreich bearbeitet worden ist. Hier ist die Funktionsdefinition die Schnittstelle. Ein Beispiel für eine Anfrage-Antwort basierte Kommunikationsart ist Representational-State-Transfer (REST) über das Hypertext-Transfer-Protocol (HTTP).

Die Vorteile dieser Kommunikationsart sind die Einfachheit und das direkte Feedback über den Erfolg einer Anfrage, dass der anfragende Dienst über die Antwort erhält. Als Nachteil ist aufzuführen, dass der anfragende Dienst abhängig von der Bearbeitungsgeschwindigkeit der Gegenseite ist. Dies kann dazu führen, dass er einen Timout bekommt, wenn das Warten auf die Antwort zu lange dauert. (Wolff, 2018)

Nachrichten basierte Kommunikation

Die Nachrichten basierte Kommunikation basiert auf dem Publish-Subscribe-Entwurfsmuster. Hierbei können Dienste Nachrichten veröffentlichen, die von mehreren anderen Diensten abonniert werden können. Hier ist also die veröffentlichte Nachricht die Schnittstelle und die Abhängigkeitsbeziehung ist entgegengesetzt zur Nachrichtenflussrichtung

Fowler (2006) unterscheidet zwei Typen von Nachrichten. Ein Event informiert über ein vollendetes und somit vergangenes Ereignis, zum Beispiel „Benutzer angelegt“ oder „Zahlung erhalten“. Ein Command (auf Deutsch „Befehl“) hingegen ist eine Aufforderung eine Aktion auszuführen.

Der große Vorteil Nachrichten basierter Kommunikation ist die Unabhängigkeit des Senders vom Bearbeiter einer Nachricht und somit auch von dessen Bearbei-

tungsgeschwindigkeit. Außerdem können weitere Empfänger einer Nachricht sehr einfach hinzugefügt werden. Allerdings wird zusätzlich ein Nachrichtensystem benötigt, das die veröffentlichten Nachrichten an die entsprechenden Abonnenten weiterleitet. Des Weiteren ist nicht mehr klar ersichtlich, welcher Dienst eine bestimmte Nachricht veröffentlicht und welche Dienste eine bestimmte Nachricht abonniert haben. Zusätzlich kommt hinzu, dass ein Dienst kein direktes Feedback über den Erfolg einer Aktion, die eine Nachricht auslöst, bekommt.

3.1.6 Event-Sourcing

Event-Sourcing bezeichnet eine alternative Form der Zustandsspeicherung. Hierbei wird nicht der Zustand selbst gespeichert, sondern die Events bzw. Nachrichten, die zu einer Zustandsänderung geführt haben. Der aktuelle Zustand lässt sich so immer aus allen bisherigen Events rekonstruieren. Ein Event-Store speichert alle Events und da Events per Definition nicht geändert werden können, ist es nur möglich neue Events anzufügen oder vorhandene Events abzufragen. Da die Erstellung des aktuellen Zustandes bei sehr vielen Events sehr lange dauert, können in regelmäßigen Abständen sogenannte Snapshots mit dem jeweils aktuellen Zustand generiert und abgespeichert werden. Mit dem letzten Snapshot und allen neuen Events kann so relativ effizient der aktuelle Zustand aufgebaut werden. (Wolff, 2018)

3.2 Abhängigkeitsgraphen

Die Abhängigkeiten zwischen den einzelnen Microservices können mit einem gerichteten Graphen modelliert und visualisiert werden. Die Microservices werden dabei durch die Knoten dargestellt und die gerichteten Kanten zwischen den Knoten beschreiben die Richtung der Abhängigkeiten. Eine Veranschaulichung eines Abhängigkeitsgraphen ist in Abbildung 3.2 zu sehen, die die Abhängigkeiten zwischen den Diensten A, B und C zeigt. Dienst A ist abhängig von den Diensten B und C, Dienst B ist abhängig vom Dienst C und Dienst C ist von keinem anderen Dienst abhängig. So kann bei einem Dienstausfall leicht der Einfluss auf andere Dienste bestimmt werden. Im Beispiel würde ein Ausfall des Dienstes B nur den Dienst A beeinträchtigen, da Dienst C nicht abhängig von Dienst B ist. (Gaidels & Kirikova, 2020)

Ein Abhängigkeitsgraph sollte idealerweise automatisch generiert werden können, damit er immer dem aktuellen Stand der Softwarearchitektur entspricht. Folgende Tools können beispielsweise aus Infrastruktur- oder Tracing-Daten einen Abhän-

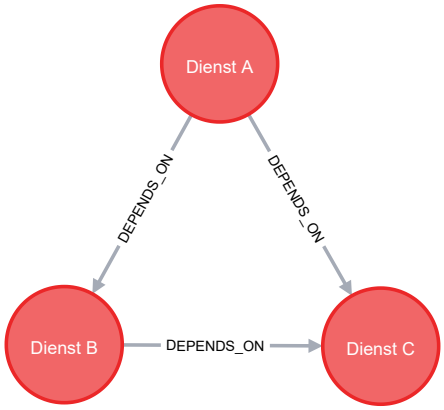


Abbildung 3.2: Abhängigkeitsgraph

gigkeitsgraph generieren: AWS X-Ray[3], Datadog[4], SPEKT8[5], Netflix Vizceral[6] und Mercator[7].

So können nicht nur Visualisierungen erstellt werden, sondern mit Graphalgorithmen können weiterführende Informationen über eine Microservice basierte Softwarearchitektur gewonnen werden (Gaidels & Kirikova, 2020; Watt, 2019):

- Zyklische Abhängigkeiten können mit Hilfe von Zyklenerkennungsalgorithmen erkannt werden.
- Ein Centrality-Algorithmus kann bestimmen, wie wichtig ein Dienst im gesamten System ist.
- Stark zusammenhängende Komponenten, die auf eine schlechte Aufteilung der Microservices hindeuten, können mit einem Cluster-Algorithmus gefunden werden.

3.3 Verteilte Systeme

Da ein auf Microservices basiertes System ein verteiltes System ist, werden nun die Grundlagen über verteilte Systeme dargestellt. „Ein verteiltes System ist eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes kohärentes System erscheinen“ (Tanenbaum & van Steen, 2007). Aus dieser Definition ergeben sich zwei wesentlichen Eigenschaften von verteilten Systemen. Zum einen

[3]https://aws.amazon.com/de/xray
[4]https://www.datadoghq.com
[5]https://github.com/spekt8/spekt8
[6]https://netflixtechblog.com/vizceral-open-source-acc0c32113fe
[7]https://github.com/LendingClub/mercator

bestehen sie aus mehreren unabhängigen Komponenten, die in Microservice basierten Architekturen die einzelnen Microservices sind. Zum anderen soll ein Benutzer nicht bemerken, dass das System aus mehreren Computern aufgebaut ist. Dies führt dazu, dass die einzelnen Komponenten untereinander bestimmte Informationen und Daten austauschen müssen, damit jede Komponente den gleichen Zustand nach außen präsentieren kann. Die unabhängigen Komponenten müssen also in gewisser Art und Weise zusammenarbeiten, um diese Eigenschaften zu erfüllen. (Tanenbaum & van Steen, 2007)

Verteilte Systeme haben eine höhere Komplexität wegen zusätzlicher Fehlerquellen, die in einem monolithischen System nicht auftreten können. Einzelne Computer können unabhängig von den anderen ausfallen, sodass Teile des Systems weiterhin funktionieren und andere nicht. Außerdem erfolgt die Kommunikation zwischen den Computern über ein Netzwerk, welches ebenfalls verschiedene Schwierigkeiten mit sich bringt. Die Netzwerkverbindung zwischen zwei Computern kann unabhängig von den anderen Verbindungen ausfallen. Es kann auch passieren, dass das Netzwerk durch Fehler in mehrere Teile aufgetrennt wird, sodass nur noch Computer im eigenen Teil erreicht werden können, aber Computer in anderen Teilen nicht mehr. Dies wird als Netzwerk-Partition bezeichnet. Zusätzlich ist die Bandbreite der Verbindung begrenzt und auch die Latenz ist nicht Null. Alle diese zusätzlichen Fehlerquellen führen dazu, dass verteilte Systeme komplexer als monolithische Systeme sind. (Tanenbaum & van Steen, 2007)

In den folgenden Abschnitten werden weitere charakteristische Merkmale und Methoden zur Behandlung von Fehlern in verteilten Systemen erläutert.

3.3.1 Konsistenzgarantie Modelle

Die oben beschriebene Eigenschaft, dass ein verteiltes System für den Benutzer wie ein einzelnes System erscheinen muss, wird oft auch als Konsistenz bezeichnet. Das System ist dann stark konsistent, wenn jede Komponente dem Benutzer einen Zustand präsentiert, der zu den Zuständen der anderen Komponenten konsistent ist. Dies bedeutet, dass ein Benutzer nach der Bestätigung über den Erfolg einer Änderung sofort diese Änderung an allen Computer des verteilten Systems sehen muss. Da starke Konsistenz nur sehr schwer zu erreichen ist, definieren Tanenbaum und van Steen (2007) und Kingsbury (2020) weitere verschiedene schwächere Konsistenzgarantien. Die schwächste Konsistenz, die in verteilten Systemen von Bedeutung ist, ist die sogenannte „eventual consistency“ (auf Deutsch „letztendliche Konsistenz“). Diese garantiert, dass ein Benutzer nach der Bestätigung über den Erfolg einer Änderung die Auswirkung dieser Änderung nicht sofort, sondern nur irgendwann in der Zukunft auch an den anderen Komponenten sehen muss.

Neben der Konsistenz ist für verteilte Systeme von Bedeutung, ob Netzwerkteilausfälle toleriert werden können und ob das System verfügbar ist, sodass es von Benutzern verwendet werden kann. Mit Hilfe dieser drei Eigenschaften definiert Brewer (2000) das sogenannte CAP-Theorem. Es besagt, dass ein verteiltes System nur zwei der drei folgenden Eigenschaften garantieren kann:

- Starke Konsistenz (C von Englisch „consistency“)
- Verfügbarkeit (A von Englisch „availability“)
- Netzwerk-Partition Toleranz (P von Englisch „partition tolerant“)

Da es in einem verteilten System immer Teilausfalle geben kann, folgt aus der ursprünglichen Interpretation, dass man sich eigentlich nur zwischen starker Konsistenz und Verfügbarkeit entscheiden kann. Allerdings hat sich mit der Zeit gezeigt, dass das CAP-Theorem in der Theorie zwar richtig ist, aber die Anwendung in der Praxis aus verschiedenen Gründen schwierig ist. Unter normalen Umständen, wenn keine Netzwerk-Partitionen oder keine Ausfälle existieren, können sowohl starke Konsistenz als auch Verfügbarkeit garantiert werden. Das heißt man muss sich nur im Fehlerfall zwischen starker Konsistenz und Verfügbarkeit entscheiden. Des Weiteren muss diese Entscheidung nicht einmalig global für das gesamte System getroffen werden, sondern kann für jede Interaktion und deren Fehlerbehandlung individuell getroffen werden. Für das gesamte System kann also meist keine Aussage darüber getroffen werden, ob es starke Konsistenz oder Verfügbarkeit garantiert. Zusätzlich kommt hinzu, dass zwischen starker Konsistenz oder Verfügbarkeit auch diverse Zwischenstufen möglich sind. Bei diesen Zwischenstufen wird eine höhere Verfügbarkeit aber im Gegenzug nur durch eine schwächere Konsistenz garantiert. In vielen realen Systemen ist es in der Regel kein Problem eine schwächere Konsistenzgarantie auszuwählen und so eine höhere Verfügbarkeit zu erreichen. Dies hat außerdem den Vorteil, dass schwächere Konsistenzgarantien leichter zu implementieren sind. (Brewer, 2012)

3.3.2 Aufrufsemantiken

In verteilten Systemen kommunizieren die einzelnen Komponenten über das Netzwerk. Da die Aufrufe verschiedenen Fehlerquellen ausgesetzt sind, gibt es verschiedene Aufrufsemantiken, die garantieren wie mit Fehlern umgegangen werden muss. Damit diese Garantien eingehalten werden können, werden Timeouts, wiederholtes Senden von Nachrichten und die Filterung von Duplikaten eingesetzt, um Fehler zu tolerieren. Es gibt folgende Aufrufsemantiken für das Tripel Anfrage, Aktion und Antwort:

- *maybe*: Die Anfrage wird genau einmal gesendet und abgebrochen nach einem Timeout.

- *at most once*: Die Anfrage wird wiederholt gesendet, wenn Timeouts eintreten. Nur die erste Anfrage, die den Server erreicht, wird ausgeführt. Alle weiteren Anfragen werden mit der ursprünglichen Antwort beantwortet.
- *exactly once*: Dies entspricht dem Idealfall, dass genau eine Anfrage gesendet wird, die entsprechende Aktion wird einmal ausgeführt und eine Antwort wird zurückgegeben. Diese Semantik ist im verteilten Fehlerfall nicht erreichbar.
- *at least once*: Die Anfrage wird wiederholt gesendet, wenn Timeouts eintreten. Jede Anfrage wird auf der Serverseite bearbeitet und jeweils die entsprechende Antwort zurückgesendet.
- *last of many*: Dies ist eine Erweiterung zu *at least once* mit der Einschränkung, dass der Client nur die Antwort akzeptiert, die zu seiner neuesten Anfrage gehört.

Bei *at least once* und *last of many* ist zu beachten, dass eine Aktion auch mehrmals ausgeführt werden kann. Damit keine Konsistenzprobleme durch die Wiederholung auftreten, sollte die Aktion idempotent sein. Denn die mehrmalige Ausführung einer idempotenten Aktion muss denselben Effekt wie die einmalige Ausführung haben. (Distler, 2018)

3.3.3 Replikation und Synchronisation

Damit es in verteilten Systemen bei Ausfällen und Fehlern nicht zu Datenverlusten kommt, müssen Daten redundant auf mehreren Computern abgespeichert werden. Dieses mehrfache Speichern von Daten auf unterschiedlichen Computern wird als Replikation bezeichnet. Somit können Computerausfälle und die damit verbundenen Datenverluste durch die anderen Computer kompensiert werden. Die gewählte Replikationsart bestimmt maßgeblich welche Konsistenzgarantie ein System gewährleisten kann. Zum Beispiel muss bei starker Konsistenz eine Änderung auf allen Replikaten durchgeführt werden, bevor dem Nutzer der Erfolg der Aktion bestätigt werden darf. Replikation erhöht nicht nur die Zuverlässigkeit von verteilten Systemen, sondern kann auch die Performance von lesenden Anfragen erhöhen. Lesende Anfragen für einen replizierten Datensatz können von allen Replikaten beantwortet werden, sodass ein deutlich höherer Durchsatz erzielt werden kann. Schreibende Anfragen sind in der Regel aufwendiger, da mehrere schreibende Anfragen auf unterschiedliche Replikate erst verteilt und somit synchronisiert werden müssen, damit keine Inkonsistenzen entstehen. Zudem ist es wichtig, dass Änderungen auf allen Replikaten in der gleichen Reihenfolge ausgeführt werden, wenn starke Konsistenz garantiert werden soll. Hierfür müssen sich die Replikate absprechen und auf eine Reihenfolge einigen. Solche Mechanismen

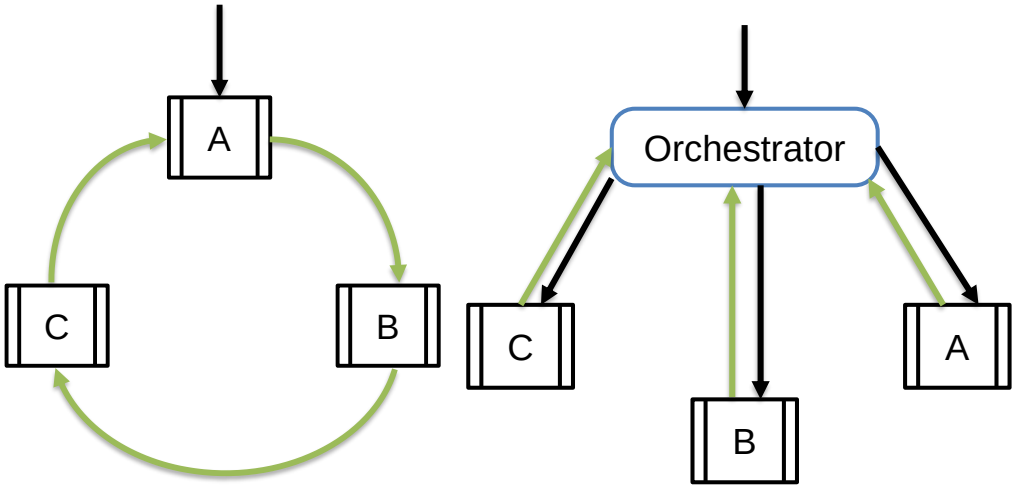
können zum Beispiel mit Hilfe von Vektoruhren oder Anführerwahl umgesetzt werden. (Distler, 2018)

3.4 Verteilte Transaktionen

In verteilten Systemen und Microservice basierten Architekturen werden oft verteilte Transaktionen benötigt, die mehrere Dienste überspannen. Dies ist ein Unterschied zu lokalen Transaktionen, die nur einen Dienst umspannen und die Transaktionseigenschaften einfach garantieren können. Hierbei ist unter anderem eine Anforderung, dass eine Aktion entweder von allen Beteiligten oder von keinem Beteiligten ausgeführt wird. Da in verteilten Systemen das Netzwerk oder Computer zu beliebiger Zeit ausfallen können, ist diese Anforderung mit den bisher beschriebenen Kommunikations- und Fehlertoleranzmechanismen nicht zu erfüllen. Deswegen werden nun drei Möglichkeiten vorgestellt verteilte Transaktionen umzusetzen.

Two-Phase-Commit Beim Two-Phase-Commit wird die Ausführung der Aktion in zwei Phasen unterteilt. In der ersten Phase wird die Änderung vom Koordinator an alle beteiligten Dienste gesendet. Jeder einzelne Dienst muss nun entweder bestätigen, dass er die Änderung lokal durchführen kann, oder ein Abbruchsignal an den Koordinator senden. Der Koordinator sammelt alle Ergebnisse ein und veranlasst das Anwenden der Änderung, wenn alle Dienste der Änderung zugestimmt haben. Falls mindestens ein Dienst nicht zugestimmt hat, wird die Transaktion abgebrochen. Newman (2020) empfiehlt den Two-Phase-Commit nicht in Microservice basierten Architekten einzusetzen, da es ein blockierendes Protokoll ist und somit nicht gut skalierbar ist. (Tanenbaum & van Steen, 2007)

Saga-Entwurfsmuster Beim Saga-Entwurfsmuster wird eine verteilte Transaktion mit Hilfe einer Sequenz von lokalen Transaktionen und Nachrichten basierter Kommunikation durchgeführt. Wenn ein Dienst eine lokale Transaktion durchgeführt hat, veröffentlicht er eine Nachricht, um die nächste lokale Transaktion bei einem anderen Dienst zu starten. Falls eine lokale Transaktion fehlschlägt, wird eine entsprechende Fehler-Nachricht veröffentlicht und für die bereits durchgeführten lokalen Transaktionen werden kompensierende Transaktionen ausgeführt, um die Änderungen rückgängig zu machen. Eine Saga kann entweder mittels Choreografie oder Orchestrierung umgesetzt werden. Bei der Choreografie basierten Saga veröffentlichen die Dienste Events, die die anderen beteiligten Dienste konsumieren. Bei der Orchestrierung gibt es einen zentralen Orchestrator, der den Diensten mitteilt, wann diese ihre lokale Transaktion ausführen sollen. In der Abbildung 3.3 sind diese zwei Möglichkeiten dargestellt. (Newman, 2020; Richardson, 2020)



(a) Saga mittels Choreografie **(b)** Saga mittels Orchestrierung

Abbildung 3.3: Darstellung einer Saga zwischen den Diensten A, B, und C mittels Choreografie (a) und Orchestrierung (b)

Transactional-Outbox-Entwurfsmuster In Microservices mit Event basierter Kommunikation ist es typisch, dass sowohl Änderungen in einer Datenbank durchgeführt werden als auch eine entsprechende Nachricht bzw. ein Event veröffentlicht wird. Um zu garantieren, dass immer beide Aktionen durchgeführt werden und im Fehlerfall keine der beiden, wird eine verteilte Transaktion benötigt, die den Microservice, die Datenbank und das Nachrichtensystem überspannt. In diesem Fall kann das Transactional-Outbox-Entwurfsmuster verwendet werden, um die verteilte Transaktion zu realisieren. Hierzu wird die zu veröffentlichende Nachricht in eine sogenannte „Outbox"-Datenbanktabelle mit der lokalen Transaktion, die die eigentlichen Änderungen in der Datenbank durchführt, eingefügt. Ein zusätzlicher Dienst, der sogenannte „Message-Relay", liest die eingefügten Einträge aus der Outbox-Tabelle und veröffentlicht die entsprechenden Nachrichten. So können mit dem Transactional-Outbox-Entwurfsmuster, welches in Abbildung 3.4 schematisch dargestellt ist, in einer Datenbanktransaktion sowohl die Daten geändert als auch eine entsprechende Nachricht erstellt werden. (Morling, 2019; Richardson, 2020)

Der Message-Relay kann die neuen Einträge der Outbox-Tabelle entweder mittels Polling abfragen oder das Transaktionsprotokoll der Datenbank verwenden. Beim Polling wird in regelmäßigen Intervallen eine normale Datenbankabfrage auf der Outbox-Tabelle durchgeführt, um die neuen Einträge bzw. Nachrichten zu ermitteln. Das Transaktionsprotokoll speichert alle Änderungen, die eine Datenbank ausgeführt hat. Es wird eigentlich verwendet, um den Zustand der Datenbank wiederherzustellen, wenn zum Beispiel ein Fehler aufgetreten ist. In vielen Datenbanken können auch externe Dienste das Transaktionsprotokoll abfragen oder

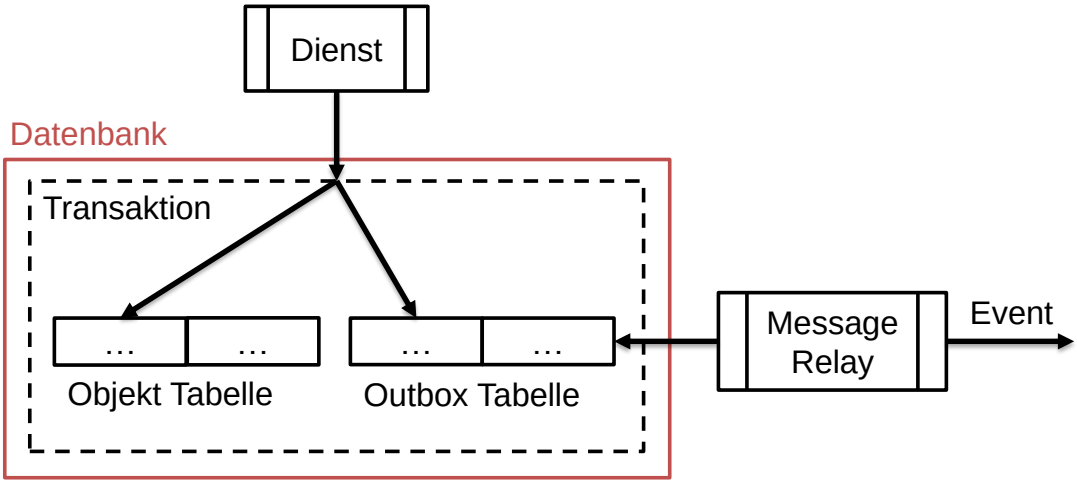


Abbildung 3.4: Transactional-Outbox-Entwurfsmuster
Quelle: in Anlehnung an Richardson (2020)

sich registrieren, um über neue Einträge in der Outbox-Tabelle informiert zu werden. (Richardson, 2020)

3.5 Chaos-Engineering

Bisher wurden viele unterschiedliche Maßnahmen vorgestellt, um verteilte Systeme fehlertoleranter und widerstandsfähiger zu machen. Nun soll beschrieben werden, wie die Wirksamkeit solcher einzelnen Maßnahmen in Bezug auf das gesamte System verifiziert und getestet werden kann. Netflix (Izrailevsky & Tseitlin, 2011), Google (Krishnan, 2012), Microsoft (Nakama, 2015), Amazon (Robbins, Krishnan, Allspaw & Limoncelli, 2012) und Facebook (Veeraraghavan et al., 2018) verwenden hierfür alle sehr ähnliche Tests. Dabei werden bewusst Fehler, wie der Ausfall von Servern, Netzwerkkomponenten oder ganzen Rechenzentren, in die Produktionsumgebung eingestreut. So kann verifiziert werden, dass solche Fehler auch im Ernstfall toleriert werden können. Hierbei geht es nicht nur darum, dass die Fehler softwareseitig und durch Redundanzen toleriert werden können, sondern auch darum firmeninterne Prozesse und Pläne zur Systemwiederherstellung zu überprüfen. Ein solches Vorgehen wird als Chaos-Engineering bezeichnet. Basiri et al. (2016) beschreiben Chaos-Engineering als das Experimentieren an einem System, um Vertrauen in die Fähigkeit des Systems aufzubauen, außergewöhnlichen Ereignissen im Produktionsbetrieb standzuhalten. Außerdem haben sie folgende vier Prinzipien identifiziert, die Chaos-Engineering definieren:

1. **Hypothesen zum normalen Verhalten aufbauen** Als erstes ist es wichtig ein Verständnis über das normale Verhalten des Systems zu erlangen.

Hierzu ist es von Vorteil verschiedene Metriken zu definieren, mit denen sich ein normales Verhalten von einem Fehlverhalten unterscheiden lässt. Die Metriken sollten nur Verhalten beschreiben, das von außen sichtbar ist, denn ein Nutzer des Systems sollte so wenig wie möglich von einem Fehler beeinträchtigt werden. Deswegen sind Metriken, wie CPU-Auslastung oder Anfragelatenz, nicht geeignet. Basiri et al., 2016 beschreiben zum Beispiel, dass Netflix unter anderem die Metrik „Stream Starts pro Sekunde“ verwendet. Im Idealfall sollte es also keine Veränderung dieser Metriken während Chaos-Engineering-Experimenten geben.

2. **Verifiziere reale Ereignisse** Bei der Auswahl von Ereignissen für ein Experiment sollten alle möglichen Ereignisse betrachtet werden. Zum Beispiel können Ereignisse, die zu vergangenen Ausfällen geführt haben, verwendet werden, um zu testen, ob das System jetzt in der Lage ist, diese Ereignisse zu tolerieren. Folgende Ereignisse werden häufig beim Chaos-Engineering eingesetzt:

   - Terminierung einer Instanz eines Dienstes
   - Erhöhung der Latenz bei Anfragen zwischen zwei Diensten
   - Verlust von Netzwerkanfragen
   - Ausfall von gesamten Rechenzentren

3. **Experimente im Produktionsbetrieb ausführen** Im ersten Moment erscheint es nicht sinnvoll, diese Experimente im Produktionsbetrieb durchzuführen, da Benutzer und Kunden durch einen potenziellen Ausfall des Systems beeinträchtigt werden können. Allerdings ist es bei der Größe heutiger verteilter Systeme oft nicht mehr möglich, das komplette System in einer separaten Testumgebung zu starten. Außerdem wird es immer Unterschiede zwischen der realen und der Testumgebung geben. Simulierte Benutzer und echte Benutzer werden sich nie identisch verhalten und auch die DNS-Konfiguration wird immer Unterschiede aufweisen. Aus diesen Gründen sollten Chaos-Engineering-Experimente möglichst immer in der Produktionsumgebung durchgeführt werden, um verlässliche Ergebnisse zu erhalten.

4. **Experimente automatisieren, damit sie ständig durchgeführt werden** Eine Automatisierung und regelmäßige Ausführung der Experimente erlaubt es mit der Zeit mehr Vertrauen in die Zuverlässigkeit des Systems zu bekommen. Kontinuierliche Tests fördern zusätzlich, dass Entwickler beim Entwerfen von neuen Diensten von Beginn an Fehlertoleranz und Ausfallsicherheit berücksichtigen.

Kapitel 4 Konzept

Im folgenden Kapitel wird das Konzept zur Überprüfung der Fehlertoleranz in Microservice basierten Softwarearchitekturen dargestellt. Hierbei werden zuerst drei unterschiedliche Abhängigkeitsgraphen vorgestellt, auf denen die Analysen basieren. Im Anschluss werden die einzelnen Analysen und Chaos-Engineering-Konzepte als Testmethode für Ausfallsicherheit erläutert. Für die gefundenen Probleme werden außerdem Lösungsmöglichkeiten aufgezeigt, wobei für anwendungsspezifische Probleme keine allgemeingültigen Lösungsvorschläge bereitgestellt werden können, da hierfür Anwendungswissen erforderlich ist.

4.1 Fehlerkultur

Bei der Entwicklung von verteilten Systemen und somit auch in Microservice basierten Architekturen ist es wichtig, dass ein Fehlerbewusstsein vorhanden ist und Fehlerbehandlungen von Beginn an berücksichtigt und eingeplant werden. Alle Entwickler müssen deswegen ein grundlegendes Verständnis von Fehlerquellen und Fehlerbehandlungsarten in den verwendeten Programmiersprachen haben. Dieses Bewusstsein ist notwendig, um eine der jeweiligen Situation entsprechende Fehlerbehandlung einzubauen, denn Fehler können in einem Programm an nahezu jeder Stelle auftreten. Dies bedeutet zum Beispiel, dass Exceptions nicht einfach nur in eine Log-Datei geschrieben werden, sondern auch entsprechende Fehlermeldungen an den Aufrufer weitergeben werden müssen, damit dieser über den Erfolg einer Aktion informiert wird. Um die Auswahl einer geeigneten Fehlerbehandlung zu vereinfachen, sollten alle Fehler in Kategorien eingeteilt und ihnen Schweregrade zugewiesen werden. Zusätzlich sollte zwischen Fehlern unterschieden werden, bei denen die Aktion wiederholt werden kann und bei denen dies nicht möglich ist. Beispielsweise wird ein Fehler aufgrund einer fehlerhaften Anfrage auch bei wiederholter Ausführung fehlschlagen und sollte deswegen nicht nochmal ausgeführt werden. Aber bei einem Fehler wegen eines Timeouts kann die Aktion durchaus nochmal ausgeführt werden.

Häufig wird in Testfällen immer nur der erfolgreiche Pfad durch ein Programm getestet. Es ist aber auch wichtig den nicht erfolgreichen Pfad zu testen, um zu überprüfen, dass eine funktionierende Fehlerbehandlung existiert.

Außerdem sollte die Steady-State-Eigenschaft (siehe Abschnitt 3.1.4) erfüllt sein, damit das Programm dauerhaft ohne Neustart ausgeführt werden kann. Hierbei ist besonders darauf zu achten, dass die Speicherkapazität nicht ausgeschöpft wird. Zum Beispiel müssen alte Log-Dateien oder alte Cache-Einträge auch zeitnah wieder gelöscht werden, damit sie nicht irgendwann den gesamten Speicherplatz belegen.

4.2 Abhängigkeitsgraph

Wie im Abschnitt 3.2 schon beschreiben, eignen sich Abhängigkeitsgraphen sehr gut, um die Abhängigkeiten zwischen den einzelnen Microservices zu visualisieren. Da ein solcher Graph auch allgemein sehr zum Verständnis eines auf Microservices basierendem System beiträgt, sollte der Graph stets den aktuellen Stand widerspiegeln. Deswegen ist es zu empfehlen ihn automatisiert generieren zu lassen, um Divergenzen zum eigentlichen System zu vermeiden.

Da Abhängigkeitsgraphen der Ausgangspunkt der Analysen zur Fehlertoleranz sind, werden im Folgenden drei Graphen auf verschiedenen konzeptionellen Ebenen dargestellt. Außerdem werden die unterschiedlichen Analysen vorgestellt, die auf diesen Graphen basieren.

4.2.1 Dienstabhängigkeitsgraph

Beschreibung

Im Dienstabhängigkeitsgraphen stellen die Kanten zwischen den Knoten eine Abhängigkeitsbeziehung dar. Hierbei beruht die Abhängigkeit rein darauf, wer die IP-Adresse und die Portnummer des anderen benötigt, um mit ihm zu kommunizieren. Diese Definition beinhaltet auch alle anderen Formen von Bezeichnern, die vor dem eigentlichen Datenaustausch in eine IP-Adresse umgewandelt werden. Solche Bezeichner können zum Beispiel der Hostname sein, der über DNS aufgelöst wird, oder eine ID, die über eine Service-Discovery aufgelöst wird. In Abbildung 4.1 ist ein solcher Graph dargestellt.

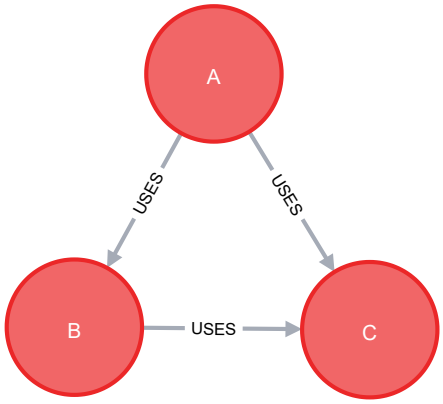


Abbildung 4.1: Beispiel eines einfachen Dienstabhängigkeitsgraphen

Analysen

Mit Hilfe eines Dienstabhängigkeitsgraphen kann bei einem Ausfall eines Dienstes sehr leicht bestimmt werden, welche anderen Dienste davon betroffen sind. Alle Dienste, die von dem ausgefallenen Dienst abhängig sind, sind somit ebenfalls von dem Ausfall betroffen oder in ihrer Funktionalität eingeschränkt. Zusätzlich legt die Abhängigkeitsbeziehung fest, in welcher Reihenfolge die Dienste gestartet werden müssen. Ein Dienst kann erst gestartet werden, wenn alle abhängigen Dienste bereits einsatzbereit sind oder die Nichterfüllung der Abhängigkeitsbeziehung durch entsprechende Maßnahmen toleriert werden kann.

Des Weiteren kann mit einem Centrality-Algorithmus bestimmt werden welche Dienste am wichtigsten sind. Der Ausfall eines wichtigeren Dienstes hat in der Regel einen größeren Einfluss auf das gesamte System als der Ausfall eines unwichtigeren Dienstes. Daher sollte ein wichtigerer Dienst ausfallsicherer sein als ein unwichtigerer Dienst. Außerdem können Cluster-Erkennungsalgorithmen helfen, stark zusammenhängende Komponenten in Graphen zu identifizieren. Dies kann auf eine ungünstige Aufteilung der Microservices hindeuten. Wenn es zum Beispiel Teilgraphen gibt, bei dem fast alle Knoten jeweils mit den anderen Knoten verbunden sind, ist es eventuell sinnvoll die beteiligen Dienste zu einem neuen Dienst zusammenzufügen oder umzustrukturieren. Die Ergebnisse von Cluster- und Centrality-Algorithmen sind sehr stark von der Architektur und dem konkreten Algorithmus abhängig. Deswegen lassen sich aus den Ergebnissen keine direkten verallgemeinerbaren Maßnahmen ableiten. Diese Ergebnisse sind aber dennoch sinnvoll, um Änderungen der eigenen Architektur zu bewerten, wenn beispielsweise ein neuer Dienst eingefügt oder ein Dienst entfernt wird.

4.2.2 Domänenabhängigkeitsgraph

Beschreibung

Der Dienstabhängigkeitsgraph beinhaltet keinerlei domänenspezifische Informationen. Außerdem ist im Dienstabhängigkeitsgraph nicht klar ersichtlich welche Domänen miteinander interagieren, da zum Beispiel bei Nachrichten basierter Kommunikation diese Information durch das Nachrichtensystem bzw. dem Message-Broker versteckt ist. Aus diesen Gründen ist es vorteilhaft zusätzlich einen Domänenabhängigkeitsgraph aufzubauen, der die Abhängigkeit zwischen den einzelnen Domänen modelliert. Hierbei wird die Abhängigkeitsbeziehung darüber definiert, welche Domäne eine Schnittstelle definiert und wer diese Schnittstelle verwendet. Somit ist die Domäne, die die Schnittstelle verwendet, abhängig von der Domäne, die die Schnittstelle bereitstellt.

Da es verschiedene Kommunikationsarten gibt, ist es sinnvoll die Art der Kommunikation in der Kante zu hinterlegen. Die typischen Kommunikationsarten in Microservice basierten Architekturen lassen sich, wie im Abschnitt 3.1.5 beschreiben, in Anfrage-Antwort und Nachrichten basierte Kommunikation aufteilen.

Anfrage-Antwort basierte Kommunikation Hierbei stellt der angefragte Dienst die Schnittstelle in Form der Funktionsdefinition bereit, weshalb der anfragende Dienst abhängig vom angefragten Dienst ist. Die Abbildung 4.2 zeigt einen solchen Domänenabhängigkeitsgraph mit zwei Domänen A und B. In dem Beispiel sendet die Domäne A die Anfrage an die Domäne B und somit ist A von B abhängig.

Abbildung 4.2: Domänenabhängigkeitsgraph mit zwei Domänen, die Anfrage-Antwort basierte Kommunikation verwenden

Nachrichten basierte Kommunikation Da bei der Nachrichten basierten Kommunikation die Schnittstelle die Form der Nachricht ist, definiert der veröffentlichende Dienst die Schnittstelle. Daher ist der konsumierende Dienst abhängig von dem veröffentlichenden Dienst und die Richtung der Abhängigkeit ist entgegengesetzt zur Nachrichtenflussrichtung.

Da Nachrichten von mehreren Diensten konsumiert werden können, ist es sinnvoll die einzelnen Nachrichtentypen explizit als jeweils eigenen Knotentyp zu model-

lieren. So wird zum Beispiel leicht ersichtlich welche Domänen eine bestimmte Nachricht konsumieren. Für jeden Nachrichtentyp wird also ein entsprechender Knoten in den Graphen eingefügt und eine Kante zu dem Dienst, der die Nachricht veröffentlicht. Außerdem wird jeweils eine Kante von den konsumierenden Diensten zu dem neuen Knoten erstellt. In der Abbildung 4.3 ist eine Nachrichten basierte Kommunikation zwischen den zwei Domänen A und B dargestellt. In diesem Beispiel wird das Event „Example Event“ von der Domäne A veröffentlicht und von der Domäne B konsumiert. Somit ist Domäne B von der Domäne A abhängig, was durch die Richtung der Kanten verdeutlicht wird.

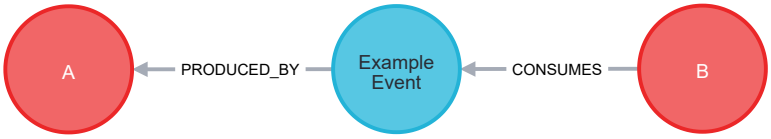


Abbildung 4.3: Domänenabhängigkeitsgraph mit zwei Domänen, die Nachrichten basierte Kommunikation verwenden

Analysen

Da die Nachrichten im Domänenabhängigkeitsgraph explizit als Knoten modelliert sind, kann leicht erkannt werden welche Nachrichten nie konsumiert oder nie veröffentlicht werden. Das Veröffentlichen von nie konsumierten Nachrichten ist nicht notwendig und sollte deswegen entfernt werden, um Ressourcen einzusparen. Wenn Nachrichten nur konsumiert aber nie veröffentlicht werden, deutet dies auf Probleme hin, denn der Konsument wird auf jeden Fall nie ausgeführt werden. Dieses Problem sollte auf jeden Fall behoben werden, indem zum Beispiel die entsprechende Nachricht an geeignete Stelle publiziert wird oder indem das Event und der Abonnent entfernt werden.

Außerdem können auf dem Domänenabhängigkeitsgraph ebenfalls Centrality- und Cluster-Algorithmen ausgeführt werden, um unvorteilhafte Aufteilungen der Domänen zu erkennen. Mit den Ergebnissen können Dienste identifiziert werden, die aufgeteilt oder zusammengefasst werden sollten.

4.2.3 Systemarchitekturgraph

Beschreibung

Für bestimmte Analysen ist nicht die Abhängigkeitsbeziehung, sondern eine Kontrollfluss oder Tracing ähnliche Beziehung wichtig. Daher wird nun der Aufbau eines Systemarchitekturgraphen beschrieben, für den ein Beispiel in Abbildung

4.4 dargestellt ist. Dieser Graph kann direkt aus dem Domänenabhängigkeitsgraph gebildet werden, indem die Richtung aller Kanten mit Nachrichten basierter Kommunikation umgedreht wird. Somit zeigen diese Kanten nicht mehr in Richtung der Abhängigkeitsbeziehung, sondern in Richtung des Nachrichtenflusses. In diesem Graph wird also der Nachrichtenfluss und die Aufrufkette der einzelnen Microservices visualisiert.

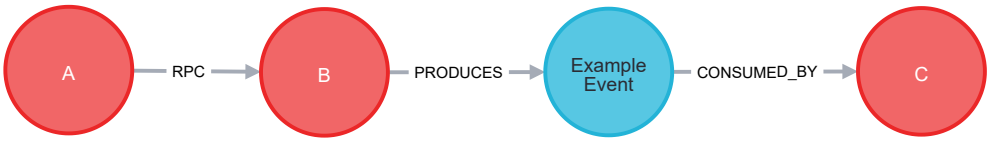


Abbildung 4.4: Beispiel eines einfachen Systemarchitekturgraphen

Analysen

Mit Hilfe des Systemarchitekturgraphen können zyklische Kommunikationsmuster gefunden werden, die ein Indikator für Architekturfehler oder Race-Conditions sein können. Deswegen sollten mit Hilfe eines Algorithmus Zyklen gesucht werden. Falls ein Zyklus gefunden wird, sollten nur die Services und Kommunikationsmuster betrachtet werden, die Teil des Zyklus sind. Für jede Interaktion im Zyklus muss nun der entsprechende Use-Case herausgesucht werden, um zu verstehen, warum diese Interaktion stattfindet. Ebenfalls von Relevanz sind Reihenfolge und Abhängigkeit der einzelnen Interaktionen. Basierend auf den gesammelten Informationen kann nun der Ursprung des Zyklus identifiziert werden und es kann bewertet werden, ob der Zyklus Probleme verursacht. Falls nötig sollten die Interaktionen mit geeigneten Maßnahmen angepasst werden.

4.3 Schreibvorgänge auf mehreren Ressourcen

Nachdem im vorherigen Abschnitt die Analysen mit Abhängigkeitsgraphen erläutert worden sind, werden im folgenden Abschnitt verschiedene Lösungen für das bei Microservices oft auftretende Problem der Schreibvorgänge auf mehreren Ressourcen dargestellt. In Microservice basierten Architekturen erfordern Datenänderungen oft Änderungen auf mehreren Ressourcen, wie zum Beispiel Datenbank, Cache, Suchindex oder Message-Broker. Sobald zwei oder mehrere Ressourcen beteiligt sind, müssen alle Schreibvorgänge auf den unterschiedlichen Ressourcen untereinander synchronisiert werden damit keine Inkonsistenzen auftreten. Die Reihenfolge der Schreibvorgänge muss dabei auf allen Ressourcen gleich sein. Wenn zum Beispiel in der Datenbank Schreibvorgang A vor B ausgeführt wird, dann muss auch auf dem Cache Schreibvorgang A vor B ausgeführt

werden. Außerdem müssen alle Schreibvorgänge immer atomar ausgeführt werden, damit entweder alle Aktionen oder keine Aktion durchgeführt wird. Um diese Eigenschaften zu gewährleisten, wird eine verteilte Transaktion benötigt, die alle beteiligten Ressourcen umfasst.

Solche Schreibvorgänge auf mehreren Ressourcen können mit Hilfe eines Dienstabhängigkeitsgraphen gefunden werden. Hierfür müssen die ausgehenden Kanten eines Dienstes betrachtet werden, die Schreibvorgänge verursachen. Wenn eine Interaktion mehr als eine dieser Kanten benötigt, dann existiert dieses Problem.

Im Folgenden werden dafür vier Lösungsmöglichkeiten vorgestellt, die auf den im Kapitel 3.4 erläuterten Möglichkeiten für eine verteilte Transaktion und Event-Sourcing (siehe Abschnitt 3.1.6) basieren.

Two-Phase-Commit

Theoretisch kann dieses Problem mit dem Two-Phase-Commit gelöst werden. Allerdings gibt es in der Praxis folgende Nachteile, weshalb die Anwendung eines Two-Phase-Commits nicht zu empfehlen ist. Zum einen setzt er voraus, dass alle beteiligten Dienste das Two-Phase-Commit-Protokoll verstehen, was bei Message-Brokern und eigenentwickelten Microservices in der Regel nicht der Fall ist. Zum anderen handelt es sich hierbei um ein blockierendes Protokoll, weshalb es nicht skalierbar ist.

Event-Sourcing

Event-Sourcing ist eine Lösung, die besonders dann gut anzuwenden ist, wenn Event basierte Kommunikation verwendet wird. Die verteilte Transaktion wird dann so umgesetzt, dass nur ein entsprechendes Event veröffentlicht wird, welches von allen anderen beteiligten Ressourcen abonniert wird. Da der Event-Store das Event persistent speichert, werden die Abonnenten auf jeden Fall irgendwann das Event zugestellt bekommen. Es kann also „eventual consistency“ garantiert werden. Außerdem werden bei Event-Sourcing die Events immer in der richtigen Reihenfolge ausgeliefert und falls ein Dienst abstürzt, können alle vergangenen Events nach dem Neustart nochmal ausgeliefert werden.

Saga-Entwurfsmuster

Wie Event-Sourcing kann das Saga-Entwurfsmuster bei Event basierter Kommunikation eingesetzt werden. Das Saga-Entwurfsmuster ist im Gegensatz zu den

anderen vorgestellten Lösungsmöglichkeiten aber auf einer höheren konzeptionellen Ebene einzuordnen. Deswegen kann es nicht so einfach angewendet werden, da für alle lokalen Transaktionen entsprechende kompensierende Transaktionen hinzugefügt werden müssen. Des Weiteren sollte beachtet werden, dass das Saga-Entwurfsmuster keine klassische Atomarität garantiert, da die einzelnen lokalen Transaktionen nacheinander ausgeführt werden. Damit trotzdem bei Fehlern keine Inkonsistenzen bestehen bleiben, werden kompensierende Transaktionen durchgeführt, um die schon erfolgten Änderungen rückgängig zu machen.

Transactional-Outbox-Entwurfsmuster

Das Transactional-Outbox-Entwurfsmuster kann dann verwendet werden, wenn sowohl eine Änderung in einer Datenbank durchgeführt als auch eine Nachricht veröffentlicht werden soll. Anstatt die Nachricht direkt zu veröffentlichen, wird sie in die Outbox-Tabelle geschrieben. Ein zusätzlicher Message-Relay-Dienst liest dann die Nachrichten aus der Outbox-Tabelle und veröffentlicht diese.

4.4 Chaos-Engineering

Mit Chaos-Engineering können verteilte Systeme sowohl auf Fehlertoleranz und Verlässlichkeit überprüft werden als auch Probleme in Bezug auf die Ausfallsicherheit gefunden werden. Dabei werden, wie in Kapitel 3.5 beschrieben, bewusst Fehler in die Produktionsumgebung eingefügt. Hierbei kann jeder Fehler, der nicht toleriert wird, die Benutzer des Systems beeinträchtigen. Aus diesem Grund sollte Chaos-Engineering in der Produktionsumgebung nur durchgeführt werden, wenn entsprechende Fehlertoleranzmechanismen, Replikation und Redundanz im System vorhanden sind. Dennoch ist es sinnvoll schon während der Entwicklung mit einer vereinfachten Form des Chaos-Engineering zu beginnen. So können Probleme in Bezug auf die Fehlertoleranz frühzeitig identifiziert und behoben werden. Zusätzlich werden die Entwickler von Anfang an geschult, auf die Ausfallsicherheit und Verlässlichkeit des Systems zu achten.

Bei der vereinfachten Form des Chaos-Engineerings werden bestimmte Aspekte an die konkreten Gegebenheiten des Systems und den Entwicklungsstand angepasst. Eine solche Anpassung ist zum Beispiel das Ausführen der Experimente in einer separaten Testumgebung, damit keine Benutzer beeinträchtigt werden. Außerdem können Art und Umfang der eingefügten Fehler angepasst werden, um die Auswirkungen der Tests einzuschränken. Ebenso ist es möglich den Datenverkehr zwischen den Microservices während der Tests einzuschränken, wenn zum Beispiel noch keine entsprechende Skalierung oder Replikation der Dienste

vorhanden ist. So können für jede Anwendung angepasste und vereinfachte Chaos-Engineering-Experimente erstellt werden. Diese sollten automatisiert werden und in den Continuous-Integration-Prozess eingebunden werden, damit die Entwickler über neue Probleme sofort informiert werden. Des Weiteren sollten die Tests mit der Zeit immer weiter ausgebaut werden, sodass irgendwann uneingeschränktes Chaos-Engineering durchgeführt wird.

Kapitel 5 Anwendung des Konzeptes am JValue ODS

5.1 JValue ODS

Der JValue Open-Data-Service (ODS) ist eine Plattform um Extract, Transform, Load (ETL)-Prozesse für Open-Data zu erstellen und auszuführen. Es können Open-Data-Quellen hinterlegt und automatisch abgefragt werden. Ebenso ist es möglich Transformationen auf den abgerufenen Daten zu definieren, um zum Beispiel das Format zu ändern oder Informationen zu extrahieren. Die Ergebnisse der Transformationen werden abgespeichert und können so für unterschiedlichste Anwendungen auf einer zentralen Plattform zur Verfügung gestellt werden. Der ODS wird von der Professur für Open-Source-Software der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) als Open-Source-Software auf GitHub[1] bereitgestellt. Aktuell befindet er sich in der Entwicklung und ist noch nicht im produktiven Einsatz.

Die Implementierung des ODS verwendet eine Microservice basierte Architektur. Die einzelnen Services kommunizieren untereinander mittels Events über das Advanced-Message-Queuing-Protocol (AMQP). Nach außen bietet jeder Microservice eine REST-Schnittstelle für Clients an. Diese wird auch von der Browser basierten Benutzeroberfläche verwendet, mit der die ETL-Prozesse definiert werden. Im Folgenden werden die einzelnen Microservices und ihre Aufgaben beschrieben.

Adapter-Service Der Adapter-Service ist in Java mit dem Spring Boot[2] Framework implementiert und verwaltet die Open-Data-Quellen. Mit Hilfe einer REST-Schnittstelle können Open-Data-Quellen definiert werden. Die Konfigurationen der Open-Data-Quellen werden in einer PostgreSQL-Datenbank[3] verwal-

[1] https://github.com/jvalue/open-data-service
[2] https://spring.io/projects/spring-boot
[3] https://www.postgresql.org

tet. Außerdem führt der Adapter auch den Import der Daten von den Open-Data-Quellen aus und speichert diese ebenfalls in der PostgreSQL-Datenbank.

Scheduler-Service Eine Datenquelle kann einen Trigger definieren, um den Import periodisch auszuführen. Der in Typescript geschriebene Scheduler-Service verwaltet diese Trigger und informiert den Adapter-Service zur entsprechenden Zeit, dass der Import einer Datenquelle ausgeführt werden soll.

Pipeline-Service Der in Typescript geschriebene Pipeline-Service ist verantwortlich für die Transformationen, die auf den importierten Daten angewendet werden können. Mit Hilfe der verfügbaren REST-Schnittstelle können Transformationen für die Datenquellen konfiguriert werden, die in einer PostgreSQL-Datenbank persistiert werden. Die Transformationen werden nach einem erfolgreichen Import ausgeführt und deren Ergebnisse werden mit einem Event an den StorageMQ-Service weitergesendet.

StorageMQ-Service Der StorageMQ-Service konsumiert die Events mit den Ergebnissen einer Transformation und speichert die Daten in einer PostgreSQL-Datenbank, der sogenannten Storage-Datenbank. Dieser Dienst ist in Typescript programmiert.

Storage-Service Mit Hilfe des Storage-Service können Nutzer über eine REST-Schnittstelle die transformierten Daten aus der Storage-Datenbank abfragen. Der Storage-Service ist keine Eigenentwicklung, sondern verwendet das Tool PostgREST[4].

Notification-Service Der Notification-Service ist in Typescript geschrieben und kann dazu verwendet werden den Benutzer zu informieren, wenn die importierten Daten erfolgreich transformiert worden sind und diese nun verwendet werden können. Aktuell können die Benachrichtigungen an die Nutzer über Webhooks, Google Firebase[5] oder Slack[6] verschickt werden. Die Benachrichtigungen können über eine REST-Schnittstelle definiert werden, deren Konfiguration in einer PostgreSQL-Datenbank abgespeichert ist.

[4]https://postgrest.org/en/v7.0.0
[5]https://firebase.google.com
[6]https://slack.com

Message-Broker Als Implementierung für AMQP wird der Message-Broker RabbitMQ[7] eingesetzt.

Edge-Router Damit die internen Details über die einzelnen Microservices für die Benutzer unsichtbar bleiben, wird der Reverse-Proxy Traefik[8] als Edge-Router eingesetzt. Dieser kann zukünftig auch als Load-Balancer verwendet werden.

5.2 Fehlerkultur

Im ersten Schritt der Anwendung des Konzeptes am JValue ODS werden, wie in Kapitel 4.1 erläutert, die Fehlerbehandlungen überprüft. Die Analyse identifizierte folgende zwei Probleme:

Als erstes sind in den Funktionen der Nachrichtenkonsumenten keine Fehlerbehandlung vorhanden. Deswegen stürzen die Microservices ab, wenn Fehler während der Bearbeitung von Nachrichten auftreten. Für dieses Problem werden jeweils Fehlerbehandlungen implementiert, die die Fehler in eine Log-Datei schreiben, sodass die Microservices bei solchen Fehlern nicht mehr ausfallen. Dies stellt allerdings noch keine perfekte Fehlerbehandlungen dar, denn solange niemand die Log-Datei kontinuierlich überwacht, bleibt der Fehler unbemerkt. Hier ist es besser die Nachrichten, die Fehler verursachen, in eine sogenannte „Dead-Letter-Queue“ einzufügen. Somit gehen diese Nachrichten nicht verloren und können durch einen entsprechenden globalen Fehlerbehandlungsmechanismus bearbeitet werden. Da die Behandlung solcher Fehler immer einen fachlichen Hintergrund haben sollte und dieser noch nicht vollständig geklärt ist, wird hier auf eine halbfertige Lösung durch die Verwendung einer Dead-Letter-Queue ohne weitere Reaktionen verzichtet.

Das zweite Problem ist, dass keine Fehlerantworten an den Aufrufer der REST-Schnittstellen zurückgesendet werden. Hierfür werden Fehlerbehandlungen eingebaut, die entsprechende Fehlerantworten erstellen und zurückgeben.

5.3 Abhängigkeitsgraph

In diesem Abschnitt werden die verschiedenen Abhängigkeitsgraphen für den JValue ODS aufgebaut und es werden jeweils die entsprechenden Analysen durchgeführt. Da der ODS noch nicht im produktiven Einsatz ist, werden alle Abhängig-

[7] https://www.rabbitmq.com
[8] https://traefik.io/traefik

keitsgraphen manuell in der Graphdatenbank Neo4j[9] modelliert. Die Graphalgorithmen können mittels Plugins direkt in Neo4j ausgeführt werden.

5.3.1 Dienstabhängigkeitsgraph

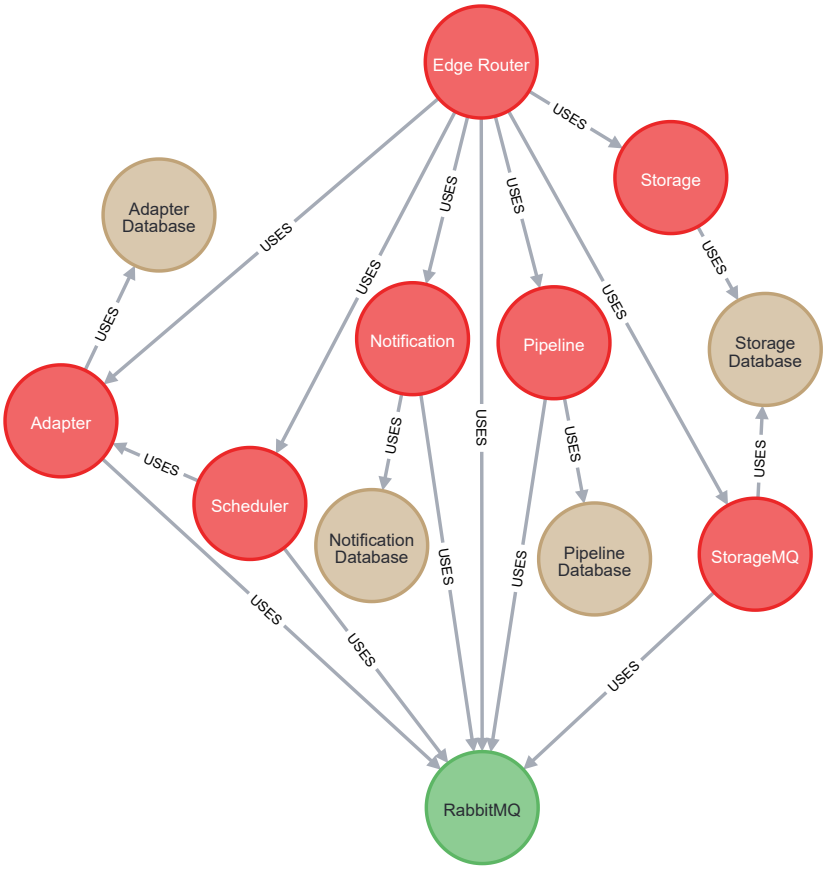


Abbildung 5.1: Dienstabhängigkeitsgraph des JValue ODS

Auf dem Dienstabhängigkeitsgraphen, der in Abbildung 5.1 dargestellt ist, wird der Page-Rank-Algorithmus ausgeführt (siehe Anhang A.1 und A.2). Dessen Ergebnisse sind in Tabelle 5.1 aufgelistet. Wie erwartet wird der Message-Broker RabbitMQ als zentralster Dienst identifiziert, denn über ihn läuft die komplette interne Kommunikation mittels Events. Diese Zentralität ist erforderlich und darf nicht beseitigt werden, damit alle Dienste weiterhin miteinander kommunizieren können. Auf Platz zwei ist die Storage-Datenbank, was vor allem daran liegt, dass sie sowohl vom Storage-Service als auch vom StorageMQ-Service verwendet wird. Dies stellt ebenso kein Problem dar, denn die Storage-Domäne ist in zwei einzelne Dienste aufgeteilt, die jeweils unterschiedliche Aufgaben haben.

[9] https://neo4j.com/product/neo4j-graph-database

Der Storage-Service führt nur lesende Operationen auf der Storage-Datenbank aus und der StorageMQ-Service nur schreibende Operationen. Bei der Interpretation der Ergebnisse ist außerdem zu beachten, welche Informationen in dem Graphen modelliert sind und wie sich dies auf die Ergebnisse der Algorithmen auswirkt. Zum Beispiel sind die Benutzer des ODS nicht explizit dargestellt und daher fehlen die eingehenden Kanten zum Edge-Router, weshalb er auch das niedrigste Page-Rank-Ergebnis hat. Allerding ist dieser nicht der unwichtigste Dienst, denn ohne ihn könnte kein Nutzer den ODS verwenden.

Dienst	Page-Rank-Ergebnis
RabbitMQ	0,56
Storage-Datenbank	0,36
Adapter-Datenbank	0,25
Adapter	0,24
Pipeline-Datenbank und Notification-Datenbank	0,22
Scheduler, StorageMQ, Storage, Pipeline und Notification	0,17
Edge-Router	0,15

Tabelle 5.1: Ergebnisse des Page-Rank-Algorithmus für den Dienstabhängigkeitsgraphen

5.3.2 Domänenabhängigkeitsgraph

In Abbildung 5.2 ist der Domänenabhängigkeitsgraph des ODS abgebildet, der mit Hilfe der Anfrage aus Anhang A.3 erstellt wird. Es ist leicht zu erkennen, dass das „Import error“ und „Transform error“ Event jeweils nur veröffentlicht, aber von keinem Dienst konsumiert werden. In einem in Produktion laufenden System sollten solche Events entfernt werden, da sie nur unnötig Ressourcen verschwenden. Da der ODS sich noch in Entwicklung befindet und eventuell in der Zukunft entsprechende Konsumenten hinzugefügt werden, werden die beiden Events noch nicht entfernt. Zum Beispiel könnte der Notification-Service noch erweitert werden, um den Benutzer über Fehler beim Import und der Pipeline-Ausführung zu informieren.

5.3.3 Systemarchitekturgraph

Der Systemarchitekturgraph des ODS kann, wie im Konzept erläutert, aus dem Domänenabhängigkeitsgraph erstellt werden und ist in Abbildung 5.3 abgebildet. Der Code zur Erstellung des Graphen ist im Anhang A.4 zu finden. Auf diesem Graphen wird ein Zyklenerkennungsalgorithmus ausgeführt (siehe Anhang A.5).

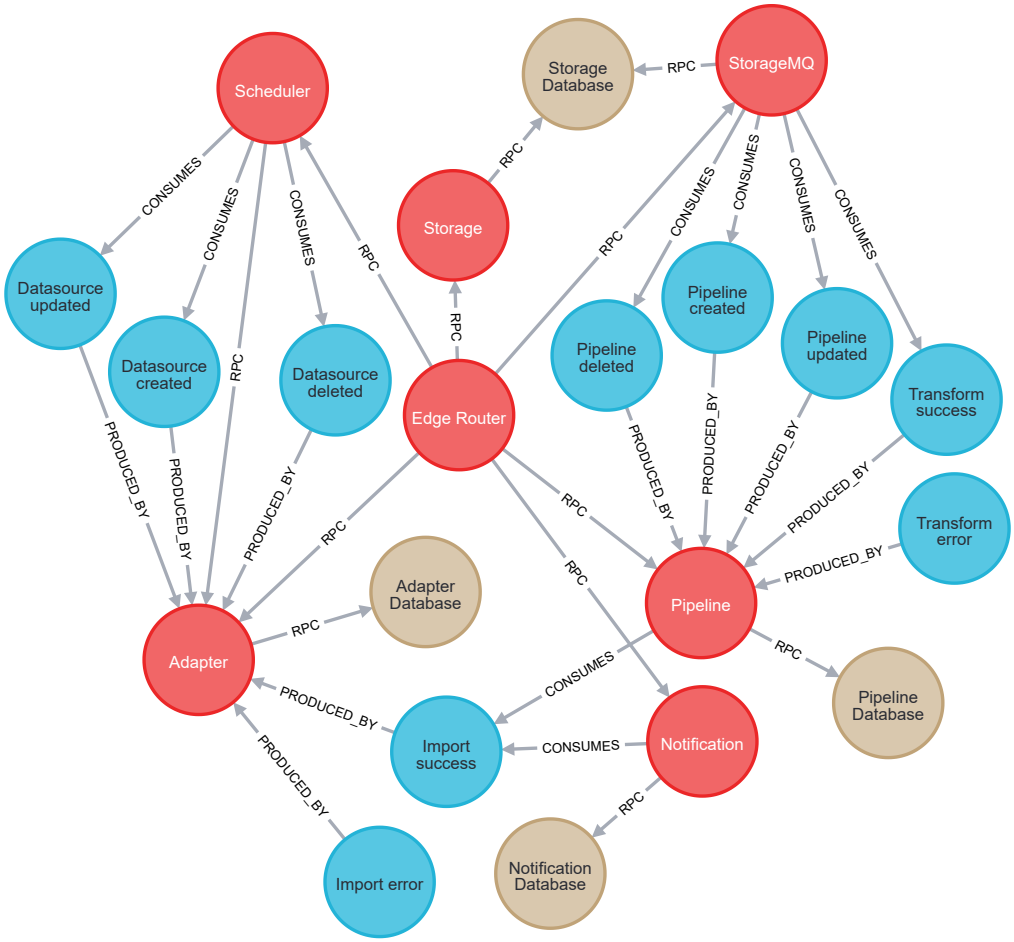


Abbildung 5.2: Domänenabhängigkeitsgraph des JValue ODS

Dieser hat einen Zyklus zwischen dem Adapter und dem Scheduler gefunden, der in Abbildung 5.4 dargestellt ist. Eine genauere Prüfung der einzelnen Interaktionen und deren Zusammenhang ergibt, dass der Zyklus durch die Initialisierung des Schedulers hervorgerufen wird. Beim Starten des Schedulers wird einmalig die Liste aller Datenquellen beim Adapter angefragt. Alle weiteren Änderungen werden dem Scheduler mittels Nachrichten mitgeteilt, indem er die entsprechenden Events konsumiert.

Problematisch ist hierbei vor allem der Übergang von der einmaligen Anfrage des aktuellen Zustandes mittels Anfrage-Antwort basierter Kommunikation zur Nachrichten basierten Übermittlung der neuen Änderungen. Damit keine Inkonsistenzen im Scheduler entstehen, müssen alle Änderungen, die nach der Zustandserzeugung im Adapter eintreten, auch als entsprechendes Event den Scheduler erreichen. Zusätzlich muss sichergestellt sein, dass Events erst bearbeitet werden, nachdem der aktuelle Zustand geladen worden ist, damit die Reihenfolge der Änderungen nicht geändert wird.

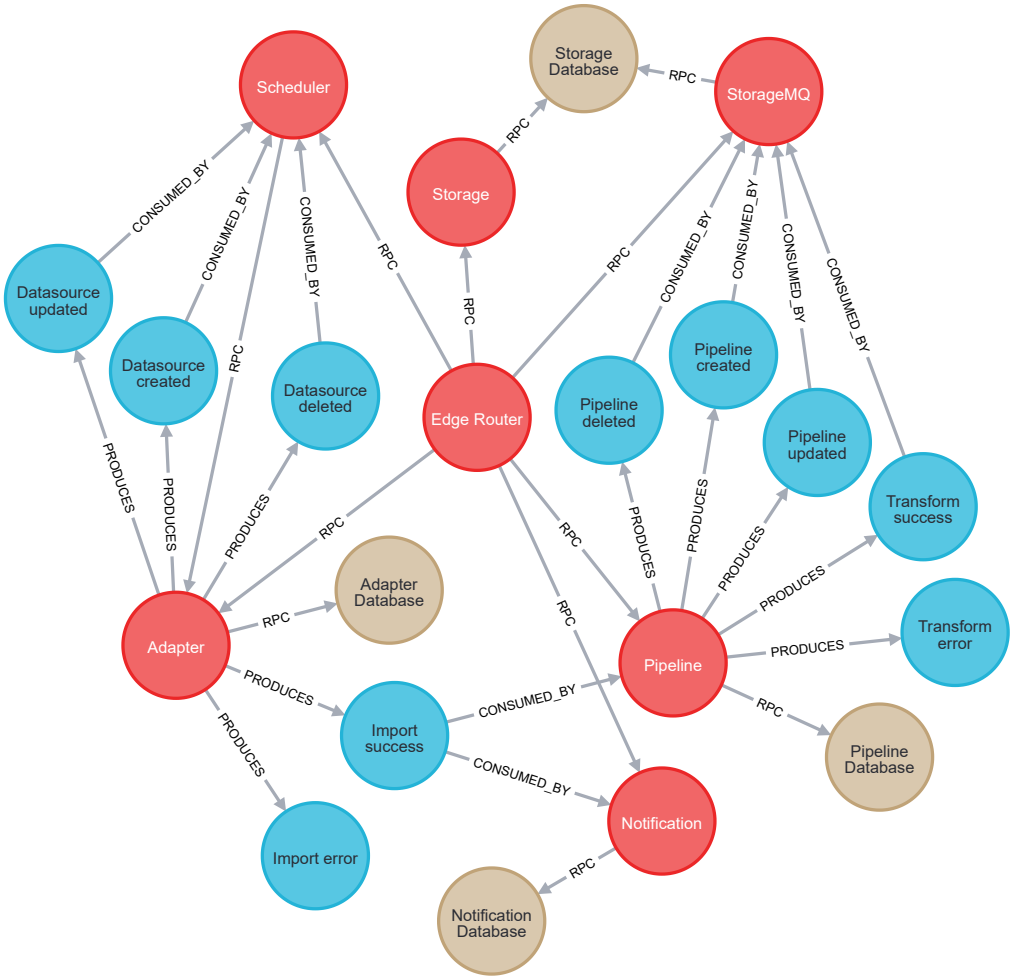


Abbildung 5.3: Systemarchitekturgraph des JValue ODS

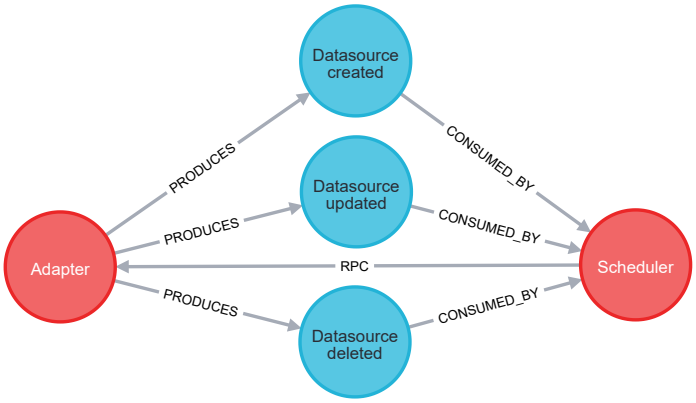


Abbildung 5.4: Ausschnitt des Systemarchitekturgraph, der den Zyklus zwischen Adapter und Scheduler zeigt

Der Scheduler abonniert zuerst die entsprechenden Events und anschließend fragt er den aktuellen Zustand beim Adapter an. Da keinerlei Synchronisation zwischen den zwei Kommunikationsarten möglich ist, können neuere Events die Antwort mit dem aktuellen Zustand überholen und so zu Inkonsistenzen führen. In der Abbildung 5.5 ist veranschaulicht, wie die Events (grüne Pfeile) der eingehenden Änderungen A, B und C (rote Pfeile) die Abfrage des aktuellen Zustandes (blaue Pfeile) überholen können. Hier ist klar zu erkennen, dass das Event von der Änderung C die Antwort mit dem aktuellen Zustand überholt. Somit werden am Ende die Änderungen in folgender Reihenfolge im Scheduler bearbeitet: B, C, A, B. Hiermit wird gezeigt, dass der Scheduler beim Starten in einen inkonsistenten Zustand gelangen kann. Im Folgenden werden vier Lösungsmöglichkeiten für dieses Problem vorgestellt.

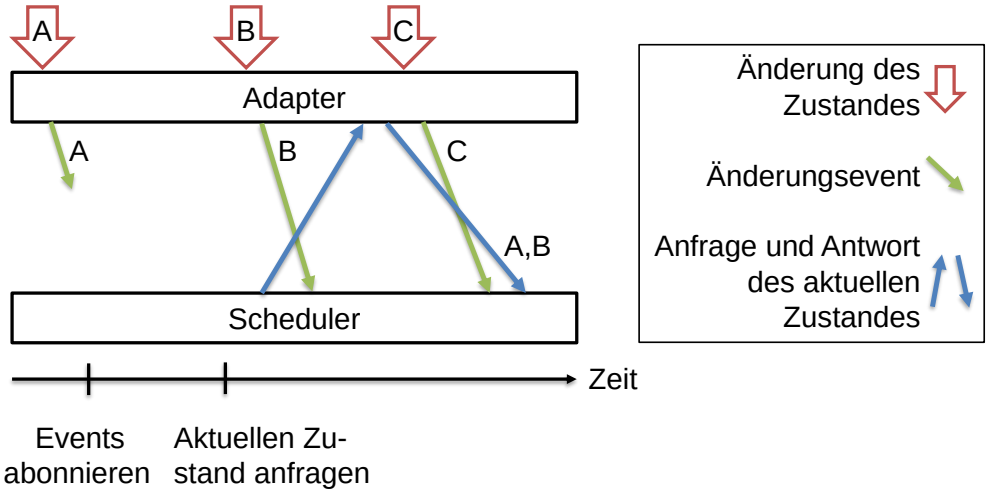


Abbildung 5.5: Inkonsistenter Zustand (B, C, A, B) bei Abonnement der Events und anschließender Zustandsabfrage

Lösung 1: Zeitstempel

Dieses Problem ist nicht ODS spezifisch, denn es kann überall auftreten, wo zuerst der gesamte Zustand geladen wird und anschließend Änderungen mittels Nachrichten übermittelt werden. Adams (2016) berichtet zum Beispiel, dass die Kommunikationsplattform Slack[10] ebenfalls dieses Problem hat, wenn die Client-Anwendung gestartet wird. Bei dieser wird auch zuerst der aktuelle Zustand beim Server abgerufen und anschließend werden die Zustandsänderungen über Nachrichten an den Client geschickt. Bei Slack wird dieses Problem so gelöst, dass neben dem Zustand auch der aktuelle Zeitstempel als Antwort zurückgesendet

[10] https://slack.com

wird. Mit Hilfe dieses Zeitstempels kann sich die Client-Anwendung als Konsument für Nachrichten registrieren. Da der Server alle Nachrichten für eine kurze Zeit persistiert, ist sichergestellt, dass keine Änderung zwischen der Abfrage des aktuellen Zustandes und der Registrierung als Konsument verloren geht. Außerdem filtert der Server mit Hilfe des Zeitstempels die Nachrichten, sodass nur die Änderungen zum Client geschickt werden, die nach der Zustandserzeugung passiert sind. Mit Hilfe eines Zeitstempels kann also das oben beschriebene Problem gelöst werden.

Beim ODS kann diese Lösung mit einem Zeitstempel aus folgenden zwei Gründen nicht umgesetzt werden. Dies liegt erstens daran, dass die Adapter-Datenbank gleichzeitige Änderungen sortiert und auch den Zustand erstellt. Aber die entsprechenden Änderungsevents werden vom Adapter veröffentlicht und somit ist eine Filterung basierend auf einem Zeitstempel nicht ohne aufwendige Synchronisation zwischen Adapter-Datenbank und dem Adapter möglich. Der zweite Grund ist, dass ein zusätzlicher Mechanismus benötigt wird, die Nachrichten für eine kurze Zeit persistent zu speichern. Damit wird sichergestellt, dass keine Änderungen verloren gehen. Da die Umsetzung dieser Lösung sehr viel zusätzliche Komplexität in das System einfügen würde, wird nach weiteren einfacheren Lösungsmöglichkeiten gesucht.

Lösung 2: Zustandsabfrage vor Eventregistrierung

In Abbildung 5.6 wird eine weitere Lösungsmöglichkeit veranschaulicht. Hier wird zuerst der aktuelle Zustand beim Adapter angefragt (blaue Pfeile) und anschließend werden die Änderungsevents (grüne Pfeile) abonniert. In der Veranschaulichung ist leicht zu erkennen, dass am Ende nur die Änderungen A und C beim Scheduler ankommen und somit wieder Inkonsistenzen auftreten können. Aus diesem Grund löst auch diese Möglichkeit das Problem nicht.

Lösung 3: Event-Sourcing

Noch eine Lösungsmöglichkeit für dieses Problem kann das in Abschnitt 3.1.6 beschriebene Event-Sourcing sein. Hier gibt es dann nicht mehr den problematischen Übergang von der Anfrage-Antwort basierten Kommunikation zur Nachrichten basierten Kommunikation, da per Definition nur noch Nachrichten verwendet werden. Der Scheduler muss sich dann beim Starten nur noch beim Event-Sourcing System registrieren und ihm werden alle vergangenen und neuen Nachrichten zugeschickt. Für diese Möglichkeit muss im ODS der Message-Broker RabbitMQ ersetzt werden, da mit ihm Event-Sourcing nicht umgesetzt werden kann. Denn RabbitMQ speichert Nachrichten nur so lange bis sie erfolgreich bearbeitet wor-

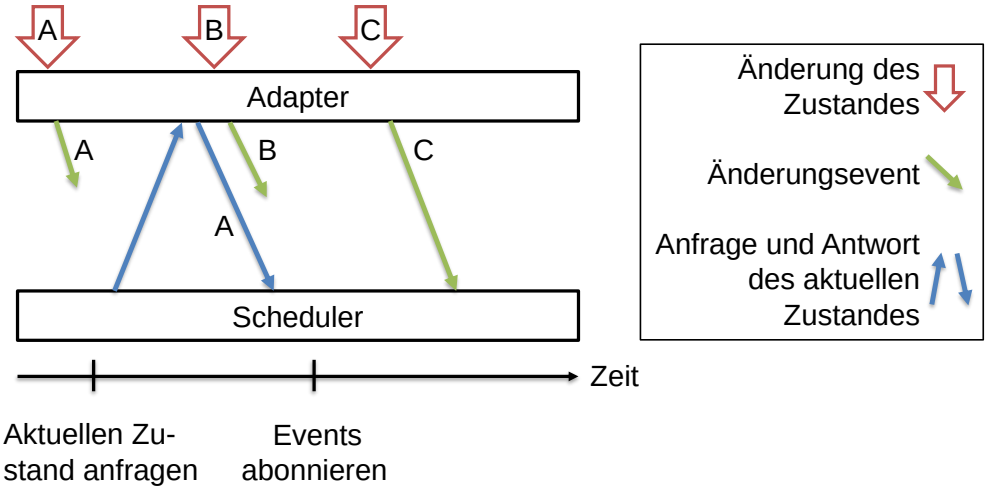


Abbildung 5.6: Inkonsistenter Zustand (A, C) bei Zustandsabfrage und anschließendem Abonnement der Events

den sind. Für Event-Sourcing müssten sie aber dauerhaft gespeichert werden. Daher wird diese Lösung auch verworfen.

Lösung 4: Eventregistrierung vor Zustandsabfrage und nachgestellte Eventbearbeitung

Eine weitere Möglichkeit ist in Abbildung 5.7 dargestellt. Hier abonniert der Scheduler zuerst die Änderungsevents, allerdings startet er noch nicht mit der Bearbeitung der Events. Anschließend lädt er den aktuellen Zustand vom Adapter (blaue Pfeile) und als letztes beginnt der Scheduler mit der Bearbeitung der Änderungsevents (grüne Pfeile). Die Bearbeitungsreihenfolge der Änderungen ist somit folgende: A, B, C, B, C. Daraus ist zu erkennen, dass es zwischenzeitlich einem inkonsistenten Zustand geben kann. Aber da die Bearbeitung der Nachrichten idempotent ist, wird am Ende wieder ein konsistenter Zustand erreicht.

Die letzte Lösung erreicht am Ende einen konsistenten Zustand und sie benötigt im Vergleich zur Zeitstempel oder Event-Sourcing Lösung am wenigsten Code-Anpassungen im ODS. Hier kann es zwar kurzeitig zu einem inkonsistenten Zustand kommen, der allerdings nur während des Starts des Scheduler auftritt und sicher wieder verlassen wird. Aus diesen Gründen wird diese Lösung ausgewählt und die kurzzeitigen Inkonsistenzen werden bewusst in Kauf genommen, da die anderen Lösungen eine höhere Komplexität in das Gesamtsystem einfügen würden. Die ausgewählte Lösung 4 (siehe Abbildung 5.7) wird im ODS implementiert.

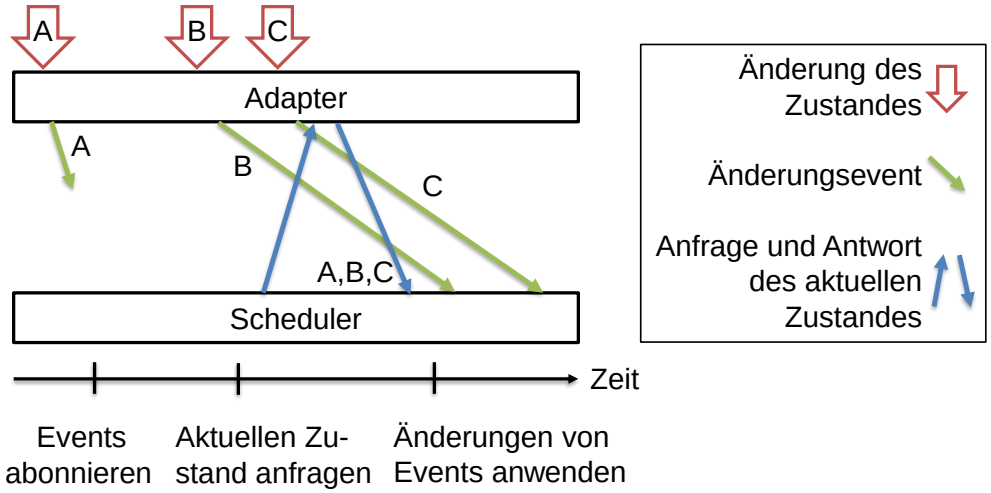


Abbildung 5.7: Kurzzeitiger inkonsistenter Zustand (A, B, C, B, C) bei Abonnement der Events vor der Zustandsabfrage und nachgestellter Eventbearbeitung

5.4 Schreibvorgänge auf mehreren Ressourcen

Als nächstes wird am JValue ODS überprüft, ob es Schreibvorgänge auf mehreren Ressourcen gibt, bei denen eine verteilte Transaktion benötigt wird. Wie im Kapitel 4.3 erläutert, können solche Interaktionen mit Hilfe des Dienstabhängigkeitsgraphen (siehe Abbildung 5.1) gefunden werden. Hierzu werden zuerst mit einer Neo4j-Abfrage, die im Anhang A.6 zu finden ist, alle Dienste ausgewählt, die mehrere andere Dienste verwenden. Da es beim Dienstabhängigkeitsgraphen keine Unterscheidung zwischen lesenden und schreibenden Aktionen gibt, werden als nächstes manuell alle Knoten mit lesenden Anfragen entfernt. So bleiben am Ende beim ODS nur noch der Adapter- und der Pipeline-Service übrig, die Schreibvorgänge auf mehrere Ressourcen ausführen. In Abbildung 5.8 sind diese beiden Dienste und die beteiligten Ressourcen dargestellt.

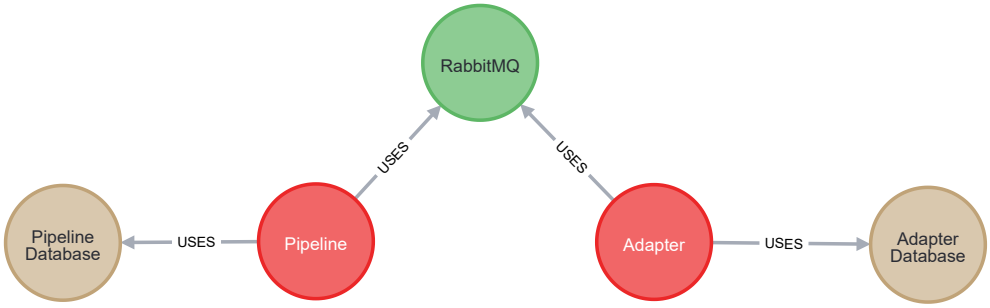


Abbildung 5.8: Ausschnitt des Dienstabhängigkeitsgraph mit den Diensten die Schreibvorgänge auf mehrere Ressourcen durchführen

Es kann somit gezeigt werden, dass das Problem mit Schreibvorgängen auf mehreren Ressourcen beim ODS im Adapter- und Pipline-Service existiert. Bei beiden Diensten werden jeweils sowohl Daten in der Datenbank geändert als auch ein entsprechendes Event veröffentlicht. Es wird folglich eine verteilte Transaktion benötigt, die den Dienst, die jeweilige Datenbank und den Message-Broker RabbitMQ überspannt. Die im Konzept (siehe Abschnitt 4.3) vorgestellten Lösungen werden nun auf ihre Anwendbarkeit im ODS untersucht.

Two-Phase-Commit Der Two-Phase-Commit eignet sich aus folgenden zwei Gründen nicht als Lösung. Zum einen müssen alle Beteiligten das Two-Phase-Commit-Protokoll implementieren, was beim Message-Broker RabbitMQ nicht der Fall ist. Zum anderen ist es, wie schon im Grundlagenkapitel beschrieben, ein blockierendes Protokoll und ist deswegen nicht skalierbar.

Event-Sourcing In der Theorie kann dieses Problem mit Event-Sourcing gelöst werden, da bereits entsprechende Änderungsevents veröffentlicht werden. Hierfür dürfen Adapter- und Pipeline-Service nicht mehr ihre Datenbank verwenden, sondern müssen ihren Zustand mit Hilfe der Events definieren. Zusätzlich muss das Nachrichtensystem um einen Event-Store erweitert werden, damit die Events dauerhaft persistent gespeichert werden können. Da dies RabbitMQ nicht unterstützt, müsste das komplette Nachrichtensystem ausgetauscht werden, wenn Event-Sourcing verwendet werden soll. Als Alternative zu RabbitMQ könnten zum Beispiel Apache Kafka[11], Axon[12] oder Eventuate[13] verwendet werden, da hiermit jeweils Event-Sourcing möglich ist.

Saga-Entwurfsmuster Das Saga-Entwurfsmuster kann hier nicht eingesetzt werden, da es auf einer höheren konzeptionellen Ebene einzuordnen ist.

Transactional-Outbox-Entwurfsmuster Das Transactional-Outbox-Entwurfsmuster passt genau zum gegebenen Problem, da sowohl eine Änderung in der Datenbank durchgeführt als auch ein entsprechendes Event veröffentlicht werden soll. Hierfür muss das Event in eine neue Outbox-Datenbanktabelle eingefügt und ein entsprechender Message-Relay-Dienst implementiert werden.

Auf Grund der Komplexität eines Umbaus auf Event-Sourcing, wird das Transactional-Outbox-Entwurfsmuster als Lösung für das Problem ausgewählt. Für die

[11] https://kafka.apache.org
[12] https://axoniq.io
[13] https://eventuate.io

Implementierung des Transactional-Outbox-Entwurfsmuster im ODS wird zuerst ein geeigneter Message-Relay-Dienst implementiert. Hierfür wird das Open-Source-Tool Debezium[14] verwendet. Debezium ist ein Change-Data-Capture (CDC) Dienst, der Änderungen in Datenbanktabellen als Nachrichten veröffentlichen kann. Es werden verschiedene Datenbanken unterstützt, unter anderem auch das im ODS verwendete PostgreSQL. Die Änderungen in der PostgreSQL-Datenbank werden mit Hilfe des Transaktionsprotokolls abgefragt und standardmäßig in Apache Kafka veröffentlicht. Debezium kann aber auch als Bibliothek in eigenen Anwendungen eingebunden werden, wenn die Änderungen nicht in Apache Kafka veröffentlicht werden sollen. Da im ODS das Nachrichtenprotokoll AMQP verwendet wird, wird mit Hilfe dieser Bibliothek ein neuer Dienst entwickelt, der die Änderungen als AMQP-Nachrichten veröffentlicht. Dieser sogenannte „Outboxer“ wird als eigenständiges Projekt auf GitHub[15] entwickelt und wird über die GitHub-Container-Registry als wiederverwendbarer Docker-Container[16] der Open-Source-Community zur Verfügung gestellt. Der Outboxer-Dienst filtert die Änderungen, sodass nur die neuen Einträge der Outbox-Tabelle veröffentlicht werden. Er kann entweder mit Hilfe einer Konfigurationsdatei oder Umgebungsvariablen konfiguriert werden und so mit der entsprechenden Datenbank und dem Message-Broker verbunden werden. Außerdem müssen der Adapter- und der Pipeline-Service angepasst werden, sodass die Events in die Outbox-Tabelle eingefügt und nicht mehr direkt veröffentlicht werden. Da die beiden Dienste jeweils ihre eigene Datenbankinstanz verwenden, muss jeweils ein eigener Outboxer-Dienst gestartet werden.

Durch den neuen Outboxer-Dienste ändern sich auch alle Abhängigkeitsgraphen. Als Beispiel ist in Abbildung 5.9 der neue Dienstabhängigkeitsgraph dargestellt und im Anhang A.7 ist der entsprechende Code zur Erstellung des Graphen zu finden. In dem Graph ist leicht ersichtlich, dass jetzt der Adapter- und der Pipeline-Service nur noch Schreibvorgänge auf ihrer jeweiligen Datenbank durchführen. Das Problem, dass es Schreibvorgänge auf mehreren Ressourcen gibt, ist somit erfolgreich behoben.

5.5 Chaos-Engineering

Als letztes werden Praktiken aus dem Chaos-Engineering auf dem JValue ODS angewendet. Da der ODS noch nicht in Produktion läuft und auch keine Skalierungs- und Replikationsmechanismen vorhanden sind, kann kein echtes Chaos-Engineering angewendet werden. Aber es können natürlich, wie im Konzept in

[14] https://debezium.io
[15] https://github.com/jvalue/outboxer-postgres2rabbitmq
[16] https://github.com/orgs/jvalue/packages/container/package/outboxer-postgres2rabbitmq

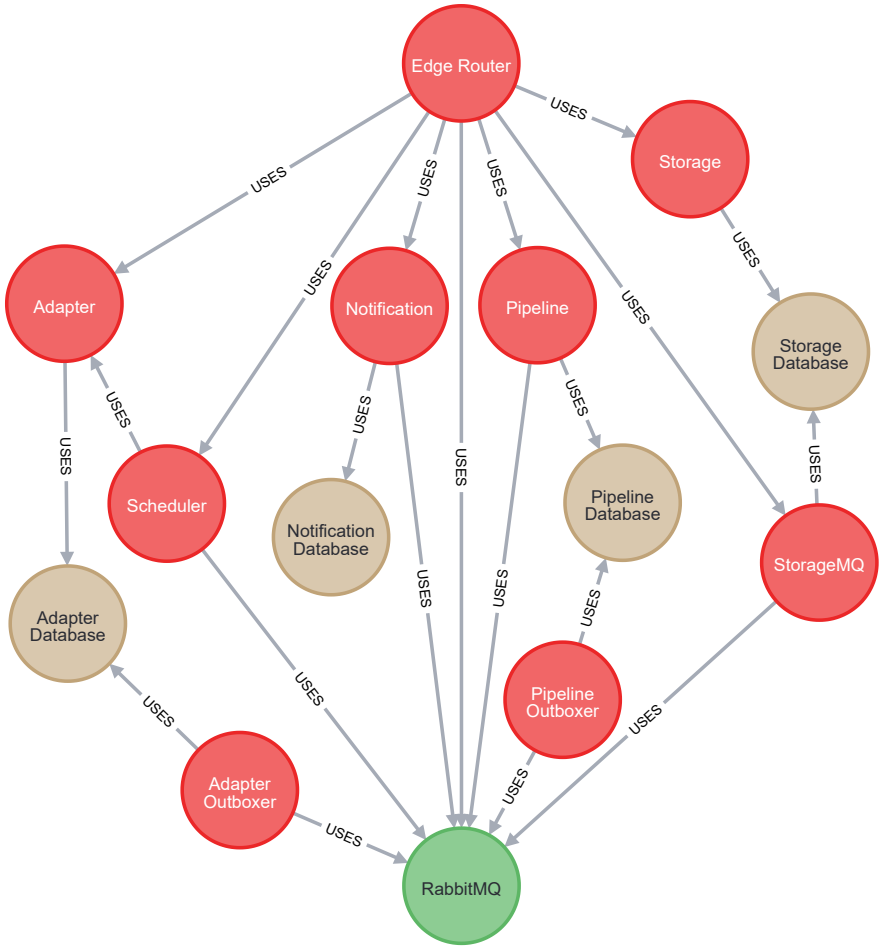


Abbildung 5.9: Dienstabhängigkeitsgraph des JValue ODS mit den Outboxer-Diensten

Abschnitt 4.4 dargestellt, vereinfachte Experimente durchgeführt werden. Daher werden für den ODS Tests implementiert, bei denen überprüft werden soll, ob Dienste gegen den kurzzeitigen Ausfall eines einzelnen Dienstes abgesichert sind. Diese Experimente werden für jeden Dienst getrennt ausgeführt, um Fehler leichter nachzuvollziehen. Alle Tests bestehen dabei jeweils aus folgenden Schritten:

1. Den gesamten ODS starten
2. Benutzerinteraktion simulieren, die zu einem Zustand im ODS führen.
3. Einen Dienst abstürzen lassen (z.B. mit `docker kill`)
4. Diesen Dienst wieder starten (z.B. mit `docker start`)
5. Überprüfen, ob der Zustand aus Schritt zwei noch vorhanden ist
6. Weitere Benutzerinteraktionen simulieren, um die gesamte Funktionalität

des ODS zu überprüfen

Bei der Ausführung ist darauf zu achten, dass während der Schritte drei und vier keine Interaktionen mit dem abgestürzten Dienst erfolgen, da dies aktuell zu nicht tolerierbaren Fehler führen würde. Außerdem ist bei den Datenbanken und dem Message-Broker darauf zu achten, dass die gespeicherten Daten auf der Festplatte nicht gelöscht werden. Denn nur mit Hilfe dieser Daten können diese Dienste den Zustand beim Starten wiederherstellen.

Die Tests werden nach dem Test-First-Ansatz erstellt, mittels GitHub-Actions automatisiert und in den Continuous-Integration-Prozess eingebunden. Dies führt zunächst zu negativen Testergebnissen, da die Lösungen noch nicht implementiert sind. So können zum einen die implementierten Lösungen dieser Arbeit auf ihre Wirksamkeit getestet werden. Zum anderen werden die Entwickler schnell informiert, falls Dienste nicht mehr gegen den Ausfall anderer Dienste abgesichert sind. Die Experimente können auch das Problem der Schreibvorgänge auf mehreren Ressourcen (siehe Abschnitt 5.4) finden und zeigen, dass die Verwendung des Transactional-Outbox-Entwurfsmusters dieses Problem behebt. Zusätzlich können mit Hilfe dieser vereinfachten Chaos-Engineering-Experimente die folgenden Probleme im ODS identifiziert und behoben werden:

Dienstausfall bei Verbindungsabbruch zur Datenbank Alle in Typescript programmierte Dienste stürzen ab, wenn die dazugehörige Datenbank abstürzt. Dieses Problem ist zurückzuführen auf eine nicht vorhandene Behandlung der Fehler, die durch einen Verbindungsabbruch verursacht werden. Wenn diese Fehler nicht behandelt werden, stürzt der gesamte Dienst ab. Dieses Problem wird bei den betroffenen Diensten jeweils durch Einfügen einer entsprechenden Fehlerbehandlung behoben.

Zustellung bereits bearbeiteter Nachrichten Nach dem Neustart eines Dienstes werden alle bereits erfolgreich bearbeiteten Nachrichten noch einmal zugestellt. Hier ist das Problem, dass keine Bestätigung über die erfolgreiche Bearbeitung der Nachricht an den Message-Broker geschickt wird. Aus diesem Grund werden die Nachrichten bei erneuter Registrierung nochmal zugestellt. Dieses Problem wird damit gelöst, dass nach der erfolgreichen Bearbeitung einer Nachricht, eine entsprechende Bestätigung an den Message-Broker RabbitMQ geschickt wird.

Nach einem Verbindungsabbruch zu RabbitMQ wird die Verbindung nicht wiederhergestellt Wenn der Message-Broker RabbitMQ neugestartet ist, bauen alle in Typescript programmierten Dienste nicht automatisch eine neue Verbindung zum Message-Broker auf. Daher werden diesen Diensten auch keine

neuen Nachrichten mehr zugesendet. Da diese Verbindung zum Message-Broker sowohl zum Veröffentlichen als auch zum Empfangen von Nachrichten verwendet wird, werden hier zwei unterschiedliche Lösungen gewählt. Beim Veröffentlichen von Nachrichten wird die Verbindung erst bei Bedarf wieder aufgebaut. Da das Konsumieren von Nachrichten im Idealfall direkt nach dem Neustart von RabbitMQ wieder funktionieren sollte, wird hier mit einem Retry gearbeitet, damit Nachrichten nicht unnötig verzögert werden. Nachdem ein Verbindungsabbruch erkannt worden ist, wird in regelmäßigen Abständen versucht die Verbindung wiederherzustellen. So ist sichergestellt, dass die Verbindung möglichst schnell wieder verwendet werden kann. Falls nach einer konfigurierbaren Zeit keine Verbindung zum Message-Broker hergestellt werden konnte, wird der Dienst beendet. So können auch Verbindungsprobleme zum Message-Broker leichter identifiziert werden.

Kapitel 6 Evaluation

In diesem Kapitel werden die Anforderungen aus Kapitel 2 hinsichtlich ihrer Erfüllung untersucht.

6.1 Allgemeine Anforderungen

Im Kapitel 4 wurde das Konzept vorgestellt, welches in Kapitel 5 am JValue ODS angewendet wurde. Die Anwendung konnte zeigen, dass mit Hilfe des Konzeptes Fehler in Bezug auf Ausfallsicherheit und Fehlertoleranz im ODS identifiziert werden konnten. Außerdem konnten die gefundenen Probleme mit den bereitgestellten Lösungsmöglichkeiten behoben werden, sodass der ODS nun fehlertoleranter und ausfallsicherer ist.

6.2 Funktionale Anforderungen

Zuerst werden die funktionalen Anforderungen sowohl für das Konzept als auch für die Anwendung am ODS hinsichtlich ihrer Erfüllung evaluiert.

6.2.1 Funktionale Anforderungen an das Konzept

Als Grundlage für das Konzept dienen drei Graphen, die auf unterschiedlichen konzeptionellen Ebenen definiert und im Abschnitt 4.2 beschrieben wurden. Der Dienstabhängigkeitsgraph und der Domänenabhängigkeitsgraph modellieren jeweils eine Abhängigkeitsbeziehung. Bei ersterem zwischen den einzelnen Diensten und bei zweiterem zwischen den einzelnen Domänen. Der Systemarchitekturgraph zeigt hingegen den Anfrage- und Nachrichtenfluss durch das gesamte System. Der Domänenabhängigkeitsgraph und der Systemarchitekturgraph unterscheiden auch zwischen Anfrage-Antwort und Nachrichten basierter Kommunikation, sodass die Anforderung auf Unterstützung von Nachrichten basierter Kommunika-

tion erfüllt ist. Für jeden Graph wurden im Abschnitt 4.2 Analysen und Muster bereitgestellt, die verschiedene Probleme in Bezug auf die Ausfallsicherheit und Fehlertoleranz identifizieren und lokalisieren können. Das Problem der Schreibvorgänge auf mehreren Ressourcen kann mit Hilfe des Dienstabhängigkeitsgraphen gefunden werden, für welches in Abschnitt 4.3 auch vier Lösungsmöglichkeiten vorgestellt wurden.

Als letzten Punkt des Konzeptes wurden im Abschnitt 4.4 Chaos-Engineering-Experimente als Testwerkzeug für Microservice basierte Softwarearchitekturen erläutert. Mit diesen Experimenten können nicht nur weitere Probleme gefunden, sondern auch die implementierten Lösungen validiert werden.

Die funktionalen Anforderungen an das Konzept konnten folglich alle erfüllt werden.

6.2.2 Funktionale Anforderungen für die Erprobung des Konzeptes am ODS

Die Anwendung des Konzeptes am JValue ODS hat verschiedene Probleme in Bezug auf die Fehlertoleranz und Ausfallsicherheit identifiziert. Aus den im Konzept bereitgestellten Lösungsvorschlägen wurde jeweils eine passende Lösung ausgewählt und implementiert. Außerdem wurden einfache Chaos-Engineering-Experimente entwickelt, mit denen die implementierten Lösungen überprüft wurden. Über einen Test-First-Ansatz konnte bestätigt werden, dass die mit den Analysen aufgedeckten Probleme im ODS behoben sind. Zusätzlich konnten die Chaos-Engineering-Experimente weitere Probleme aufdecken, die ebenfalls beseitigt wurden. Somit ist der ODS jetzt fehlertoleranter und ausfallsicherer, da Fehler sich nicht mehr auf andere Dienste ausbreiten können. Allerdings bedeutet dies nicht, dass der ODS jetzt vollständig ausfallsicher ist. Denn aktuell sind zum Beispiel noch keine Skalierungs- und Replikationsmechanismen im ODS vorhanden, die aber benötigt werden, wenn Ausfälle vollständig durch andere Instanzen kompensiert werden sollen.

Die funktionalen Anforderungen an die Erprobung des Konzeptes am ODS konnten somit erfüllt werden.

6.3 Nichtfunktionale Anforderungen

Nachdem die funktionalen Anforderungen evaluiert worden sind, werden im Folgenden die nichtfunktionalen Anforderungen hinsichtlich ihrer Erfüllung analysiert.

6.3.1 Nichtfunktionale Anforderungen an das Konzept

Das Konzept wurde nur am JValue ODS angewendet, da weitere Erprobungen an anderen Systemen mit Microservice basierten Architekturen den Rahmen dieser Arbeit sprengen würden. Inwiefern das Konzept wiederverwendbar ist, kann daher nicht vollständig evaluiert werden. Allerdings haben auch Gaidels und Kirikova (2020) und Watt (2019) über erfolgreiche Analysen von Microservice basierten Architekturen mit Hilfe von Abhängigkeitsgraphen berichtet. Deshalb ist davon auszugehen, dass die im Konzept enthaltenen Analysen mit Abhängigkeitsgraphen auch auf andere Systeme mit Microservices anwendbar sind. Das Problem mit den Schreibvorgängen auf mehreren Ressourcen (siehe Kapitel 4.3) ist kein ODS spezifisches Problem, denn Newman (2020), Morling (2019) und Richardson (2020) stellen jeweils Lösungen für dieses Problem bereit. Es ist ein allgemeines Problem, das durchaus auch in anderen Microservice basierten System gefunden werden kann.

Die leichte Anwendbarkeit des Konzeptes ist dadurch gegeben, dass das Konzept aus mehreren Schritten besteht, die nacheinander bearbeitet werden. Außerdem gibt es drei unterschiedliche Abhängigkeitsgraphen mit jeweils getrennt ausführbaren Analysen.

Die nichtfunktionalen Anforderungen an das Konzept konnten in Summe nur teilweise erfüllt werden, da die Evaluierung der Wiederverwendbarkeit mit nur einer Erprobung nicht vollständig durchgeführt werden kann.

6.3.2 Nichtfunktionale Anforderungen an die Implementierung von Lösungen am ODS

Die implementierten Lösungen für die gefundenen Probleme ändern keine Funktionalität des ODS. Es gab für die Benutzer des ODS keine Einschränkungen und Unterbrechungen. Außerdem wurde beim Problem der Schreibvorgänge auf mehreren Ressourcen bewusst keine Lösung mit Event-Sourcing ausgewählt, da dafür der Austausch des gesamten Nachrichtensystems notwendig gewesen wäre. Mit dem Transactional-Outbox-Entwurfsmuster (siehe Kapitel 5.4) wurde die Entwicklungskomplexität niedrig gehalten, da der neue Outboxer-Dienst das neue Verhalten kapselt. Generell wurden die implementierten Lösungen mit fachlichem Hintergrundwissen motiviert, wie zum Beispiel beim Zyklus zwischen Adapter und Scheduler (siehe Abschnitt 5.3.3). Rein technisch könnte dieses Problem auch mit Event-Sourcing oder einem Zeitstempel gelöst werden, allerdings wurde hier bewusst die einfachere fachliche Lösung ausgewählt.

Alle gefundenen Probleme wurden den Entwicklern des ODS in Form von Issues auf GitHub und im ODS-Developer-Meeting mitgeteilt. Die Lösungsmöglichkei-

ten wurden ebenfalls vorgestellt, diskutiert und es wurde jeweils gemeinsam eine passende Lösung ausgewählt. Die Änderungen wurden folglich abgesprochen und die anderen Beteiligten waren nicht unnötig eingeschränkt.

Die nichtfunktionalen Anforderungen an die Implementierung von Lösungen am ODS konnten somit erfüllt werden.

Kapitel 7 Fazit

Im Fazit wird neben der Zusammenfassung der Arbeit auch ein Ausblick auf weitere Themen gegeben.

7.1 Zusammenfassung

Da es sich bei Microservice basierten Softwarearchitekturen um verteilte Systeme handelt, gibt es spezifische Probleme, wie der Ausfall einzelner Dienste und des Netzwerkes, die behandelt werden müssen. Damit keine Inkonsistenzen entstehen und eine Ausbreitung von Fehlern verhindert wird, müssen entsprechende Maßnahmen zur Fehlertoleranz und Ausfallsicherheit vorhanden sein. In dieser Arbeit wurde ein Konzept erarbeitet, mit dem Microservices hinsichtlich der Fehlertoleranz und Ausfallsicherheit analysiert werden können. Dabei werden die Fehlerquellen mit Hilfe von Mustern in drei verschiedenen Abhängigkeitsgraphen identifiziert. Der Dienstabhängigkeitsgraph, Domänenabhängigkeitsgraph und Systemarchitekturgraph sind auf verschiedenen konzeptionellen Ebenen definiert, damit jeweils unterschiedliche Probleme gefunden werden können. Außerdem werden Lösungsvorschläge bereitgestellt, mit denen sich die gefundenen Probleme beheben lassen. Als letzten Schritt beinhaltet das Konzept die Beschreibung der Durchführung von Chaos-Engineering-Experimenten, um sowohl die eingebauten Maßnahmen zu überprüfen, als auch weitere Fehler finden zu können.

Das Konzept wurde am JValue ODS erprobt. Zum einen konnte so die Anwendbarkeit des Konzeptes überprüft werden und zum anderen konnten problematische Stellen in Bezug auf die Fehlertoleranz und Ausfallsicherheit im ODS identifiziert werden. Diese Schwachstellen wurden mit Hilfe der bereitgestellten Maßnahmen erfolgreich beseitigt. Am ODS wurden auch vereinfachte Chaos-Engineering-Experimente durchgeführt, mit denen nicht nur weitere Probleme aufgespürt, sondern auch die implementierten Lösungen validiert wurden.

7.2 Ausblick

Für zukünftige Optimierungen sind noch Weiterentwicklungen denkbar. Als erstes könnte die Verwendung eines Service-Mesh in das Konzept integriert werden. Ein Service-Mesh ist eine Infrastrukturschicht, mit der die Kommunikation zwischen den einzelnen Microservices beeinflusst und zentral gesteuert werden kann. Somit kann ein Service-Mesh unter anderem Funktionen, wie Service-Discovery, Lastverteilung, Retries, Circuit-Breaker und Rate-Limiting, übernehmen, die dann nicht mehr in den Microservices implementiert werden müssten. Zusätzlich sollte das Konzept an weiteren Microservice basierten Systemen erprobt werden, um die Wiederverwendbarkeit besser bewerten zu können. Außerdem kann dann mit den Ergebnissen der Erprobungen das Konzept mit neuen Analysen und Lösungsmöglichkeiten erweitert werden.

Auch beim ODS gibt es noch Potential bezüglich weiterer Verbesserungen der Fehlertoleranz. Hier ist vor allem anzuführen, dass noch keine Skalierungs- und Replikationsmechanismen vorhanden sind. Diese werden benötigt, damit der Ausfall eines Dienstes durch weitere Instanzen toleriert werden kann. Wenn diese Mechanismen eingebaut sind, können auch die Chaos-Engineering-Experimente erweitert werden. Die Fehlerbehandlung bei der Nachrichtenkonsumierung sollte auch noch verbessert werden. Hier geht es vor allem darum für jeden Nachrichtentyp zu entscheiden, wie auftretende Fehler sinnvoll auf fachlicher Ebene behandelt werden können. Eine Möglichkeit ist zum Beispiel das erneute Ausliefern der Nachricht oder das Ausführen fehlerkompensierender Aktionen (siehe auch Saga-Entwurfsmuster in Abschnitt 3.4). Empfohlen werden kann auch die Nutzung einer sogenannten „Dead-Letter-Queue“, in die fehlerverursachende Nachrichten eingefügt und durch einen entsprechenden globalen Fehlerbehandlungsmechanismus bearbeitet werden. Die Auswahl einer geeigneten Lösung sollte nicht nur hier, sondern immer hauptsächlich fachlich motiviert sein, damit die Komplexität niedrig gehalten wird und das gesamte System leicht verständlich bleibt.

Anhang A Neo4j-Abfragen

Hier werden alle verwendeten Abfragen für die Neo4j-Graphdatenbank in der Sprache Cypher[1] aufgelistet.

A.1 Dienstabhängigkeitsgraph

Die Cypher-Anfrage zur Erstellung des Dienstabhängigkeitsgraphen ist auf Grund der Länge nur in digitaler Form auf der angefügten CD vorhanden.

A.2 Page-Rank-Algorithmus

Der Page-Rank-Algorithmus wird mit Hilfe der Graph Data Science Bibliothek[2] ausgeführt. Hierzu muss zuerst mit folgendem Befehl eine Projektion erstellt werden.

```
CALL gds.graph.create('graph', 'Service', 'USES')
```

Mit Hilfe der Projektion kann nun der Page-Rank-Algorithmus ausgeführt werden.

```
CALL gds.pageRank.stream('graph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC
```

[1] https://neo4j.com/developer/cypher
[2] https://neo4j.com/docs/graph-data-science/current

A.3 Domänenabhängigkeitsgraph

Die Cypher-Anfrage zur Erstellung des Domänenabhängigkeitsgraphen ist auf Grund der Länge nur in digitaler Form auf der angefügten CD vorhanden.

A.4 Systemarchitekturgraph

Die Cypher-Anfrage zur Erstellung des Systemarchitekturgraphen ist auf Grund der Länge nur in digitaler Form auf der angefügten CD vorhanden.

A.5 Zyklenerkennungsalgorithmus

Mit Hilfe der folgenden Cypher-Anfrage können auf dem Systemarchitekturgraphen Zyklen identifiziert werden. Diese Anfrage benötigt das Plugin Awesome Procedures On Cypher (APOC)[3].

```
MATCH
  (m1)-[]->(m2), cyclePath=shortestPath((m2)-[*]->(m1))
WITH m1, nodes(cyclePath) as cycle
WHERE
  id(m1) = apoc.coll.max([node in cycle | id(node)])
RETURN m1, cycle
```

A.6 Schreibvorgänge auf mehreren Ressourcen

Das Problem der Schreibvorgänge auf mehreren Ressourcen kann mit folgender Cypher-Anfrage auf dem Dienstabhängigkeitsgraph identifiziert werden.

```
MATCH (s1:Service)-[r1:USES]->(s2:Service),
  (s1)-[r2:USES]->(s3:Service)
WHERE NOT s1.name = "Edge Router"
RETURN s1, s2, s3, r1, r2
```

Auf dem Systemarchitekturgraph kann das Problem der Schreibvorgänge auf mehreren Ressourcen mit folgender Cypher-Anfrage identifiziert werden.

[3]https://neo4j.com/labs/apoc

```
MATCH (s1:Service)-[r1:PRODUCES]->(event:Event),
  (s)-[r2:RPC]->(s2:Service)
RETURN s1, event, s2, r1, r2
```

A.7 Dienstabhängigkeitsgraph mit den Outboxer-Diensten

Die Cypher-Anfrage zur Erstellung des Systemarchitekturgraphen mit den Outboxer-Diensten ist auf Grund der Länge nur in digitaler Form auf der angefügten CD vorhanden.

Anhang B Compact Disk

Die angefügte CD beinhaltet diese Arbeit, alle Abbildungen und den Quellcode der Implementierungen.

Literaturverzeichnis

Adams, K. (2016). How Slack Works. Zugriff 10. Dezember 2020 unter https://youtu.be/WE9c9AZe-DY?t=1099

Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J. & Rosenthal, C. (2016). Chaos Engineering. *IEEE Software*, *33*(3), 35–41. doi:10.1109/ms.2016.60

Brewer, E. (2000). Towards robust distributed systems. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*. ACM Press. doi:10.1145/343477.343502

Brewer, E. (2012). CAP Twelve Years Later: How the "Rules"Have Changed. Zugriff 24. Februar 2021 unter https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Distler, T. (2018). *Lecture Notes in Distributed Systems*. Department of Computer Science 4 (Distributed Systems und Operating Systems) at Friedrich-Alexander-Universitiät Erlangen-Nürnberg.

Evans, J. (2016). Mastering Chaos - A Netflix Guide to Microservices. Zugriff 23. November 2020 unter https://www.youtube.com/watch?v=CZ3wIuvmHeM

Fowler, M. (2006). Event Collaboration. Zugriff 23. Februar 2021 unter https://martinfowler.com/eaaDev/EventCollaboration.html

Fowler, M. (2015). Microservice Trade-Offs. Zugriff 13. November 2020 unter https://martinfowler.com/articles/microservice-trade-offs.html

Fowler, M. & Lewis, J. (2014). Microservices Guide. Zugriff 20. Oktober 2020 unter https://martinfowler.com/articles/microservices.html

Gaidels, E. & Kirikova, M. (2020). Service Dependency Graph Analysis in Microservice Architecture. In R. A. Buchmann, A. Polini, B. Johansson & D. Karagiannis (Hrsg.), *Perspectives in Business Informatics Research* (S. 128–139). Springer International Publishing. doi:10.1007/978-3-030-61140-8_9

Izrailevsky, Y. & Tseitlin, A. (2011). The Netflix Simian Army. Zugriff 8. Februar 2021 unter https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J. & Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*, *35*(3), 24–35. doi:10.1109/ms.2018.2141039

Kingsbury, K. (2020). Consistency Models. Zugriff 30. November 2020 unter https://jepsen.io/consistency

Krishnan, K. (2012). Weathering the Unexpected. Zugriff 8. Februar 2021 unter https://queue.acm.org/detail.cfm?id=2371516

Loukides, M. & Swoyer, S. (2020). Microservices Adoption in 2020. Zugriff 13. November 2020 unter https://www.oreilly.com/radar/microservices-adoption-in-2020

Morling, G. (2019). Reliable Microservices Data Exchange With the Outbox Pattern. Zugriff 27. Februar 2021 unter https://debezium.io/blog/2019/02/19/reliable-microservices-data-exchange-with-the-outbox-pattern/

Nakama, H. (2015). Inside Azure Search: Chaos Engineering. Zugriff 8. Februar 2021 unter https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering

Newman, S. (2020, 30. Oktober). *Vom Monolithen zu Microservices*. Dpunkt.Verlag GmbH.

O'Hanlon, C. (2006). A Conversation with Werner Vogels. *ACM Queue*, *4*(4), 14–22. doi:10.1145/1142055.1142065

Richardson, C. (2020). A pattern language for microservices. Zugriff 27. Februar 2021 unter https://microservices.io/patterns/index.html

Robbins, J., Krishnan, K., Allspaw, J. & Limoncelli, T. (2012). Resilience Engineering: Learning to Embrace Failure. *ACM Queue*, *10*(9). Zugriff 8. Februar 2021 unter https://queue.acm.org/detail.cfm?id=2371297

Tanenbaum, A. S. & van Steen, M. (2007, 1. November). *Verteilte Systeme*. Pearson Studium.

Veeraraghavan, K., Meza, J., Michelson, S., Panneerselvam, S., Gyori, A., Chou, D., ... Xu, T. (2018). Maelstrom: Mitigating Datacenter-Level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (S. 373–389). OSDI'18. Carlsbad, CA, USA: USENIX Association.

Watt, N. (2019). Using Graph Theory and Network Science to Explore your Microservices Architecture. Zugriff 8. April 2021 unter https://youtu.be/0G5O1ffYIPI

Wolff, E. (2018, 1. August). *Microservices - Grundlagen flexibler Softwarearchitekturen*. Dpunkt.Verlag GmbH.