Möglichkeiten zur Kontrolle und Steuerung der Kommunikation von Microservices

**Bachelorarbeit**

im Studiengang  
Softwaretechnik und Medieninformatik

vorgelegt von

**Gerrit Wildermuth**Matr.-Nr.: 749442

am 28. Februar 2020   
an der Hochschule Esslingen

Erstprüfer/in: Prof. Dr. Mirko Sonntag   
Zweitprüfer/in: Prof. Dr. Steffen Schober

# Kurzfassung

Der Titel dieser Arbeit weist direkt auf den Hauptteil dieser Arbeit hin. In diesem wird sich bezüglich der Steuerung genauer mit der Thematik der Leitung und Management der Kommunikation beschäftigt, sowie auch wie diese gewährleistet werden kann. Zu jeder dieser Thematiken werden hierbei unterschiedliche Ansätze untersucht, welche sich sowohl in der Kommunikationsebene als auch der Implementierungstiefe unterscheiden und in einer Microservice Umgebung eingesetzt werden können. Vor der Überleitung in den Hauptteil setzt sich diese Arbeit zuerst mit der Thematik von Microservices im Allgemeinen, sowie der Einführung in weitere Themenbereiche der verteilten Systeme, auseinander. Dies ist für ein besseres Verständnis der folgenden Thematik wichtig.

Der Grund für die Notwendigkeit dieser Arbeit entstammt der Problematik der verteilten Systeme. Um mehr Leistung und größere Systeme zu ermöglichen wurde zu verteilten Systemen gewechselt. Dies zog allerdings auch einige Probleme nach sich, welche sich unter anderem in einer erhöhten Kommunikation auswirkten. Damit schneller auf Lastenunterschiede reagiert und weniger Ressourcen verbraucht werden, wird darum vermehrt auf Microservices gesetzt. Um hierbei einen Überblick zu bekommen welche Techniken und Bausteine dafür benötigt werden oder hilfreich sind, um Systeme zu gestalten, welche mit dieser Problematik umgehen, wurde auf diese Thematik eingegangen und unterschiedliche Möglichkeiten gesucht diese umzusetzen.

Hierbei wurde im Speziellen der Unterschied zwischen einer eher herkömmlichen Umsetzung in Java mit Spring Boot, mit einer alternativen Herangehensweise durch den Einsatz eines Service Meshes untersucht.

**Schlagwörter**: Microservice, Service Mesh, verteilte Systeme, Kommunikation.

# Abstract

This thesis is built around distributed systems and the emerging problems which come along with choosing such architecture. First it is taking a look at the theoretical background of microservices and distributed systems as well as additional systems, which are needed later on. After that the main part consists around different techniques and components which are needed or helpful in the context of microservices in distributed systems and the problems which come along with it.

For the different approaches it was specifically looked into Java with Spring Boot and a service mesh approach with Istio.

**Keywords:** microservice, distributed systems, service mesh, techniques, components.

# Inhaltsverzeichnis

Kurzfassung 2

Abstract 2

Inhaltsverzeichnis 3

Abbildungsverzeichnis 5

Tabellenverzeichnis 5

Codeverzeichnis 5

1 Einleitung 7

2 Microservices 9

2.1 Wieso Microservices 9

2.2 Monolith 9

2.2.1 Einführung 9

2.2.2 Vorteile der Monolithen 10

2.2.3 Nachteile der Monolithen 10

2.3 Microservices 11

2.3.1 Einführung 11

2.3.1 Vorteile der Microservices 12

2.3.2 Nachteile der Microservices 13

2.4 Zusammenfassung 13

3 Verteilte Systeme 15

3.1 Definition 15

3.2 Orchestrierung 15

3.3 Kommunikation in einem Verteilten System 16

3.4 Von Fehlerpotential zu Robustheit 16

3.5 Daten Beständigkeit 16

3.6 Einführung in Kubernetes 17

3.6.1 Kubernetes Komponenten 18

3.7 Vereinheitlichte Systeme 20

3.7.1 Service-Mesh 20

3.7.2 Spring 22

3.7.3 Vergleich Istio gegenüber Spring Cloud 24

3.8 Zusammenfassung 25

4 Möglichkeiten zum Management und Gewährleisten eines verteilten Systems 26

4.1 Kommunikations-Leitung 26

4.1.1 Einleitung 26

4.1.2 API Gateway 26

4.1.3 API Microgateway 28

4.2 Kommunikations - Überwachung und - Verfolgung 29

4.2.1 Einleitung 29

4.2.2 API Management 29

4.2.3 Distributed tracing 31

4.3 Gewährleistung von Services 33

4.3.1 Einleitung 33

4.3.2 Resilient4j VS Failsafe 33

4.3.3 Bulkhead 34

4.3.4 Circuit Breaker 38

4.3.5 Retry 42

4.3.6 Rate Limiting 44

4.3.7 Fallback 46

4.3.8 Zusammenfassung 48

4.4 Service discovery 48

4.4.1 Einleitung 48

4.4.2 Was ist Service discovery 49

4.4.3 Serverseitige discovery 49

4.4.4 Clientseitige discovery 50

4.4.5 Selbstregistrierungsmethode 51

4.4.6 Drittparteiregistrierungsmethode 51

4.4.7 Umsetzung 51

Zusammenfassung 55

Ausblick 57

Ehrenwörtliche Erklärung 58

5 Literaturverzeichnis 59

# Abbildungsverzeichnis

Abbildung 1 Monolith gegen Microservices Quelle: Eigene Darstellung 10

Abbildung 2: Container VS virtuelle Maschinen Vorteile der Microservices (Sebastian Eschweiler 2019) 12

Abbildung 3: Kubernetes Industrie Annahme (StackRox 2019) 15

Abbildung 4: Container deployment in Kubernetes (Ushio 2018) 17

Abbildung 5: Kubernetes Komponenten und deren Interaktion (Ushio 2018) 19

Abbildung 6: Deplyoment Yaml (Kubernetes) 19

Abbildung 7: Nutzung von Bibliotheken und die Auslagerung in Sidecars (Prinz 2019) 21

Abbildung 8: Service Mesh Architektur (Prinz 2019) 21

Abbildung 9: Spring Annahme (Paraschiv 2018) 23

Abbildung 10: API Gateway Übersicht (Richardson 11/20/2019) 27

Abbildung 11: API Gateways Backends for frontends (Richardson 11/20/2019) 28

Abbildung 12: API Management Gateway Overview (Posta 2019) 30

Abbildung 13: Distributed Tracing (Kyma) 32

Abbildung 14: Schiff Links Ohne Bulkhead (IBM 11/28/2019) 34

Abbildung 15: Schiff Rechts mit Bulkhead (IBM 11/28/2019) 34

Abbildung 16: Fehlerhafter Service ohne Bulkheads (Eigenkreation) 35

Abbildung 17: Fehlerhafter Service mit Bulkhead (Eigenkreation) 36

Abbildung 18: Implementierungswaage (Eigenkreation) 36

Abbildung 19: Circuit Breaker Zustände (Fowler 2014) 39

Abbildung 20: Retry Pattern (Microsoft) 43

Abbildung 21: Rate Limiter (Storozhuk 2018) 45

Abbildung 22: Wieso ist Service discovery notwendig (Richardson 2015) 48

Abbildung 23: Server-Side discovery Pattern (Richardson 2015) 49

Abbildung 24: Client-side discovery Pattern (Richardson 2015) 50

# Tabellenverzeichnis

Tabelle 1: Spring Cloud vs. Istio 1 (Eigenkreation) 24

Tabelle 2: Spring Cloud vs. Istio 2 (Eigenkreation) 24

Tabelle 3: Service Mesh VS API Management (Eigenkreation) 31

# Codeverzeichnis

[Codeteil 1: Bulkhead Serverseitige Implementierung in Java mit resilience4j in Plain Java 37](#_Toc33705074)

[Codeteil 2: Bulkhead Fehlermeldung wenn der Threadpool vollgelaufen ist. 38](#_Toc33705075)

[Codeteil 3: Resilient4j mit Spring Boot 2. 38](#_Toc33705076)

[Codeteil 4: Methode welche mit Resilient Annotationen des Circuit Breakers versehen ist. 41](#_Toc33705077)

[Codeteil 5: Circuit breaker Einstellungen in der application.properties 41](#_Toc33705078)

[Codeteil 6: Cicuit breaker durch eine destination Rule in Istio 42](#_Toc33705079)

[Codeteil 7: Retries Einstellungen 43](#_Toc33705080)

[Codeteil 8: Method mit Resilient4j Retry 43](#_Toc33705081)

[Codeteil 9: Retries durch einen VirtualService in Istio 44](#_Toc33705082)

[Codeteil 10: Methode mit Rate Limiter durch Resilence4j 45](#_Toc33705083)

[Codeteil 11:Einstellungen des rate Rimiter von Resilience4j 46](#_Toc33705084)

[Codeteil 8: Methode mit Fallback 47](#_Toc33705085)

[Codeteil 13: Aufzurufende Fallback Methode 47](#_Toc33705086)

[Codeteil 10: Service discovery Server 52](#_Toc33705087)

[Codeteil 11: Service discovery Client 52](#_Toc33705088)

[Codeteil 16: Kubernetes Namensraum 53](#_Toc33705089)

[Codeteil 17: Kubernetes Deployment 53](#_Toc33705090)

[Codeteil 18: Kubernetes Service welcher die App verfügbar macht 54](#_Toc33705091)

[Codeteil 19: Kubernetes DNS Name 54](#_Toc33705092)

# Einleitung

Diese Arbeit beschäftigt sich im Besonderen mit der Kommunikation zwischen bzw. von Microservices und möchte hierbei unterschiedliche Möglichkeiten für Techniken und Bausteine im Kontext der verteilten Systeme untersuchen. Zuallererst wird sich dafür mit theoretischen Hintergründen von Microservices, verteilten Systemen und weiteren benötigten Grundlagen befasst, welche für ein besseres Verständnis später benötigt werden. Die Notwendigkeit dieser Arbeit entsteht dem Problem, welches fast unweigerlich auftaucht, wenn man sich mit verteilten Systemen beschäftigt. Durch ein verteiltes und vernetztes System kann bei weitem mehr Kommunikation entstehen als dies bei bisherigen Systemen der Fall war. Dieser Kommunikationsanstieg kann dazu führen, dass deutlich mehr Fehler auftreten, sich deutlich mehr um den Kommunikationsfluss gekümmert werden muss, es nicht mehr möglich ist manuell einzelne Serviceadressen weiter zu verteilen und ein Management einzelner APIs immer schwieriger wird. Es gibt im Umfeld der verteilten Systeme durchaus noch weitere Probleme, um welche sich gekümmert werden könnte, jedoch werden diese im Umfang dieser Arbeit nicht näher betrachtet.

Es sollen hier unterschiedliche Möglichkeiten aufgezeigt werden, um ein Robusteres verteiltes System zu ermöglichen und mit Fehlern wie Kettenreaktionen, Ressourcen Erschöpfung, Netzwerkunterbrechungen oder Methoden welche zufällig einen Fehler werfen, entsprechend umzugehen.

Bevor allerdings Kommunikationsfehler auftreten können benötigt es eine Kommunikation. Um diese Kommunikation zu ermöglichen, benötigt man Kenntnis über die Services, im Speziellen über ihren Aufenthaltsort. Damit wir diesen in einem sich möglicherweise schnell verändernden System, noch finden können, braucht es eine dynamische Lösung zum Lösen dieses dynamischen Problems.

Nachdem Kenntnis über den Aufenthaltsort der Services besteht, muss der Kommunikationsfluss in dem verteilten System wieder ermöglicht bzw. hergestellt werden. Hier müssen Bausteine zur Anwendung kommen, welche helfen sich mit diesem Problem zu befassen und es ermöglichen, anstehende Anfragen durch das System durchzuleiten, um sie genau dort hinzuführen, wo sie hingehören.

Nach der Wiederherstellung des Kommunikationsfluss und der wiederhergestellten Zugänglichkeit der APIs, wird es daraufhin wichtig zu kontrollieren, wer Zugriff auf das System hat. Sowohl Nutzer als auch Entwicklern benötigen eigene Möglichkeiten sich mit dem System zu verbinden und mit diesem zu arbeiten, unbewusst oder bewusst.

Um diese Probleme zu bändigen, sollen sowohl unterschiedliche Möglichkeiten auf der Kommunikationsebene als auch in der Implementierungstiefe geboten werden. Dabei wird hier lediglich auf die Programmiersprache Java eingegangen, um den Rahmen nicht zu sprengen. Die hierbei Verwendung findenden Umgebungen sind zum einen Spring Boot/Cloud, als auch das Service Mesh Istio.

Im Folgenden wird eine Einführung in Microservices gegeben, mit einer Gegenüberstellung zu einem Monolithischen Ansatz, welcher dabei helfen soll herauszufinden, ob ein Microservice Ansatz der richtige für das entsprechende Problem ist, sowie einige weitere Probleme aufzeigt welche hiermit einhergehen können.

# Microservices

Der weitere Verlauf befasst sich mit Microservices und erklärt woher diese kommen und worauf sie aufgebaut sind. Zudem erfolgt eine Einführung in die Thematik von Microservices. Weiter wird eine Gegenüberstellung mit einem Monolithischen System gegeben wodurch einige weitere Vor- und Nachteile dieser beiden Systeme veranschaulicht werden. Der Leser soll dadurch in die Lage versetzt werden, eine Entscheidung zu treffen, welches eher das richtige System für seine Anwendung ist.

## Wieso Microservices

Der Ursprung von Microservices entstammt aus Problemen, welche mit bisherigen monolithischen Architekturen nicht mehr zu bewältigen waren. “The many things that you would like to see happening in a good software environment couldn’t be done anymore”. (Vogels 30/6/2006). Diese Systeme konnten ab einem Zeitpunkt/einer Größe nicht länger wachsen/sich entwickeln. “It couldn’t evolve anymore. The parts that needed to scale independently were tied into sharing resources with other unknown code paths.” (Vogels 30/6/2006). Die monolithische Architektur ist in diesem Fall an ihre Grenzen gekommen und eine neue Architektur musste die alte ersetzten, eine Architektur, welche es möglich macht, Softwarekomponenten schnell, unabhängig und vor allem isoliert zu bauen und zu skalieren. Microservice bietet eine solche Architektur, welche es ermöglicht, diese Punkte umzusetzen und setzt sich damit immer mehr als Trendsetter durch. „Microservices have become mainstream in the enterprise.“ (Dimensional Research 2018).

## Monolith

### Einführung

Ein Monolith ist im Deutschen ein einzelner Stein. In diesem Fall spiegelt es ein zusammenhängendes Softwaresystem wider, welches alle Komponenten in sich vereinigt. Diese Komponenten, wie in Abbildung 1 zu sehen ist, können in unterschiedliche Teile aufgeteilt sein, das System wird aber als einheitliches Paket veröffentlicht. Diese eng verbundene Bauweise ist ihr Segen wie auch ihr Fluch zugleich. Die einzelnen Teile des Systems können dadurch direkt lokal kommunizieren und so einen sehr hohen Datendurchsatz erreichen. Auf der anderen Seite kann nur die einzelne Maschine, auf der das System veröffentlicht wurde, weiter verstärkt werden, nicht aber das System auf mehrere Standorte verteilt und individuell skaliert werden. Außerdem muss bei jeder Änderung auch das gesamte System neu veröffentlicht werden. Teilbereiche separat zu veröffentlichen ist hierbei nicht möglich.

Abbildung 1 Monolith gegen Microservices Quelle: Eigene Darstellung

### Vorteile der Monolithen

* Der größte Vorteil eines Monolithen ist die nahe Verknüpfung der einzelnen Bestandteile. Somit kann auf weitläufige Kommunikation verzichtet und einiges an Fehlern sowie Arbeitsaufwand vermieden werden. Dadurch ist es möglich schneller zu Ergebnissen zu kommen, was sich vor allem zu Beginn eines Projektes bemerkbar macht, da keine zusätzlichen Systeme benötigt werden um z.B. das System zu veröffentlichen.
* Durch die zusammenhängende Architektur gestaltet es sich einfacher, das System auf genau einem Server zum Laufen zu bringen. Dies ist dadurch möglich, dass keine weiteren Verbindungen, Ports oder ähnliches festgelegt werden müssen, da alles auf einem und demselben System läuft und keine weiteren verwendet werden können.
* Die Datenbank liegt an einem Punkt, wodurch Datenkonsistenz deutlich einfacher bzw. überhaupt erreicht werden kann.

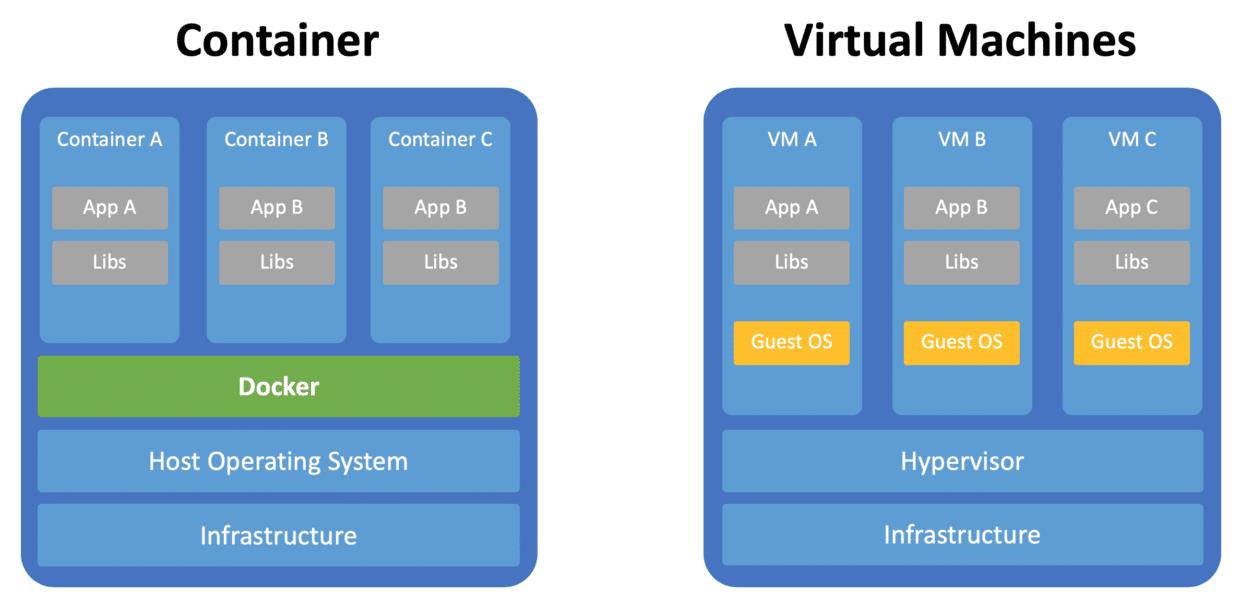
### Nachteile der Monolithen

* Jede Änderung des Systems, egal ob klein oder groß, hat zur Ursache, dass das gesamte System neu bereitgestellt werden muss. Dies spielt keine große Rolle, solange die Anwendung noch klein ist, doch je größer sie wird, umso länger dauert dieser Prozess.
* Ein weiteres Problem von zu groß werdenden, zusammenhängenden Systemen ist, dass durch fehlende Isolierung jede Änderung ungeahnte Folgen haben kann. Dadurch wird es schwieriger neue Technologien in ein bestehendes System zu integrieren.
* Je größer die Anwendung wird, also umso mehr Zeilen Code sie enthält und zusätzliche Features implementiert werden, umso wertvoller werden Entwickler, die sich bereits mit dem System auskennen. Ebenso kann es auch immer schwieriger für neue Entwickler werden einzusteigen und eigene Beiträge beizusteuern.
* Da die einzelnen Teilbereiche stark miteinander verbunden sind und oft an vielen Orten gleichzeitig gearbeitet werden muss, können sich zu viele Entwickler leicht in die Quere kommen. Dies hat zur Folge, dass früher eine maximale Anzahl an Entwicklern erreicht wird.

## Microservices

### Einführung

Verschiedene Bereiche der Anwendung werden aufgeteilt und in eigene Services ausgelagert. Jeder Service kommuniziert über Schnittstellen mit anderen Services bzw. bietet diese an. Jeder Service ist eine eigenständige leichte, also schnell startende und wenig Platz benötigende, Anwendung. Für eine bessere Performance und Isolierung wird häufig eine einzelne Datenbank (siehe Abbildung 1), in viele einzelne Datenbanken aufgeteilt. Es hat sich hierbei etabliert das Microservices für gewöhnlich in Containern auftreten. „The best choice for running a microservices application architecture is application containers.” (Golden 2/5/2017) Container bilden hierbei ein Gehäuse für eine entsprechende Anwendung mit all ihren Abhängigkeiten, Laufzeitumgebungen, Systemwerkzeugen, Systembibliotheken und Einstellungen (Abbildung 2). Die Container bilden isolierte Bereiche ab, mit eigenen Namenräumen, Benutzern und Ressourcen. Sie stellen hierbei allerdings keine vollwertigen Virtuellen Maschinen dar und besitzen kein eigenes Betriebssystem oder einen eigenen Kernel. Dies ist allerdings auch gleichzeitig der Grund für ihre schlankere Bauweise, wie man im Vergleich in Abbildung 2 sehen kann. „Just from an efficiency perspective, containers are a far better choice for a microservices architecture than are VMs.” (Golden 2/5/2017)

Abbildung 2: Container VS virtuelle Maschinen Vorteile der Microservices (Sebastian Eschweiler 2019)

### Vorteile der Microservices

* Die Anwendung in kleinere handlichere Teilbereiche zu zerteilen hilft dabei den Code leichter zu verstehen und sich besser auf das Wesentliche zu konzentrieren, zu entwickeln und zu pflegen.
* Je größer die Anwendung wird, umso eher können ganze Teams auf einzelne Services angesetzt werden und diese entwickeln, anstatt sich um die gesamte Anwendung kümmern zu müssen.
* Vorausgesetzt die Services sind nicht eng miteinander verzahnt, können Entwicklerteams freier entscheiden, was für Technologien, Frameworks oder Sprachen sie benutzten.
* Die jeweiligen Bereitstellungen können durch eine Isolierung vom restlichen System getrennt werden, was es einfacher macht Änderungen durchzuführen. Tests bleiben ebenso auf kleinere Services begrenzt, wodurch sie schneller veröffentlicht werden können.
* Ein Skalieren der Services wird möglich und durch die ressourcensparende Bauweise kostensparender. Im Gegensatz zum monolithischen System, wo jeweils das einzelne System verstärkt wird, werden im Microservice Kontext nur diejenigen skaliert, welche wirklich benötigt werden, ohne unnötige Ressourcen zu verschwenden.
* Durch die starke unabhängige Modularität können Updates je nach Service vollständig unbemerkt durchgeführt werden und es ist nicht mehr notwendig das gesamte System abzuschalten. Durch diesen beschleunigten Prozess können Entwickler auch schneller auf Änderungen oder Fehler reagieren.
* Wenn Fehler in Services keine Kettenreaktion herbeiführen, wird eine höhere Robustheit erreicht und das gesamte System kann stabiler laufen.
* Durch die Verwendung von Service discovery können automatisch skalierte Services sofort verwendet werden.
* Durch die Verwendung von Management Komponenten wird es einfacher die eigenen APIs zu beherrschen.
* Services können, durch die sprachliche Trennung, einfach in anderen Projekten wiederverwendet werden.

### Nachteile der Microservices

* Da Microservices in der Regel in einem verteilten System zuhause und dadurch über das Netzwerk getrennt sind, wird es bei größeren Systemen deutlich schwieriger Fehler zu finden. Dies wird noch verstärkt, wenn die Services von unterschiedlichen Teams entwickelt werden.
* Wenn Services gebaut werden kann die Unterteilung der logischen Prozesse zu weit gehen. Dadurch werden diese zu sehr auseinander gezerrt, was die Nützlichkeit der Verteilung überschreitet und die Prozesse unnötig verlangsamt.
* Die Notwendigkeit der Verwendung von Docker oder Kubernetes, um das verteilte System zu steuern bzw. bereitzustellen, möchten manche nicht auf sich nehmen, da sie z.B. nicht noch weitere Technologien benutzen möchten oder das Know-how dafür fehlt.
* Mit vielen Abhängigkeiten zwischen den Services kann es herausfordernd werden einen Ende zu Ende Test durchzuführen.
* Kommunikation zwischen Services, kann sehr teuer werden, wenn z.B. an der falschen Stelle auf synchrone, anstatt asynchrone Kommunikation gesetzt wird, was zu lange blockenden Anfragen führen kann.
* Datenbankanfragen, können dazu führen das zusätzliche Datenbanken angesprochen werden müssen, welche zu anderen Services gehören. Dies kann zu Problemen führen (wenn diese z. B. mehrfach kaskadierend sind).

## Zusammenfassung

Die vorangegangenen Erläuterungen sollten einen Überblick über folgende Aspekte, bezüglich Microservices, verschafft haben: Erstens sollte deutlich geworden sein, was Microservices sind und wie sich ihre Verbindung mit Containern darstellt. Des Weiteren sollte nun ein Verständnis dafür gewonnen sein was ein Service und dessen Äquivalent in einem Monolithen ist, sowie, was es mit der Monolithischen Architektur auf sich hat und wo die Unterschiede zur Microservice Architektur liegen. Darauf beruhend sollte es nun möglich sein eine fundierte Entscheidung zwischen diesen beiden Architekturen zu treffen, um den passenderen Architekturstil für das jeweilige Projekt zu finden. Zudem sollte ein Überblick der Vor- und Nachteile sowie der Risiken der beiden Architekturstile ersichtlich sein.

Im Folgenden werden einige Aspekte von verteilten Systemen erläutert und eine tiefere theoretische Grundlage für die weitere Arbeit gegeben, wobei besonderer Bezug auf Microservices genommen und die genauere Problematik in verteilten Systemen aufgezeigt werden.

# Verteilte Systeme

## Definition

Nach (Tanenbaum 2006) ist ein verteiltes System ein Zusammenschluss unabhängiger Computer, welche dem Nutzer als ein einziges, zusammenhängendes Systems erscheinen. In diesem Fall wird dieses verteilte System als ein zusammenhängendes System aus Microservices betrachtet. Dem entsprechend trifft die vorherige Definition der Microservices hier auf die einzelnen Komponenten/Computer zu.

## Orchestrierung

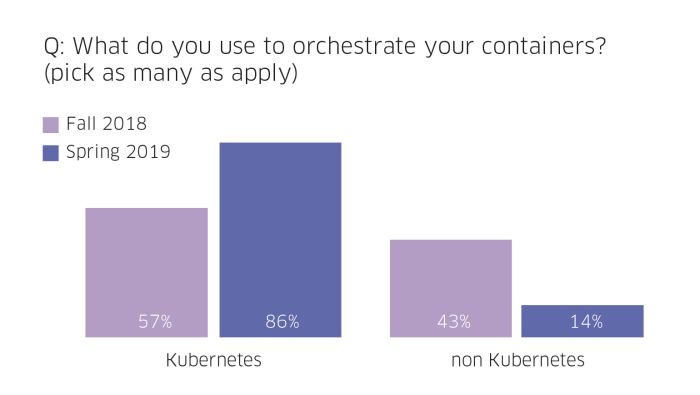
Bei der Verwirklichung eines verteilten Systems mit einer Microservice Architektur ist ein gutes Management der dadurch entstehenden Schnittstellen, aufgrund der deutlich größeren Anzahl an Services, unabdingbar. Durch die Verwendung von Containern und Autoskalierungsoptionen können Serviceinstanzen sich deutlich schneller vermehren. Für diese zum einen deutlich größere Anzahl an Services und zum anderen nun große Anzahl an Instanzen, werden neue Werkzeuge benötigt, um dieses Wachstum zu beherrschen und möglichst zu automatisieren. Diese Werkzeuge werden in der Fachsprache Orchestratoren genannt und werden dafür benutzt diese Services automatisiert zu konfigurieren, zu koordinieren und zu managen,

Abbildung 3: Kubernetes Industrie Annahme (StackRox 2019)

Wobei sie sich natürlich in der Funktionalität und Handhabung unterscheiden. Hinsichtlich der vorliegenden Untersuchungen wird auf die Orchestrierungs Plattform Kubernetes zurückgegriffen, da diese zum Zeitpunkt der Thesis-Erarbeitung die am weitesten verbreitete Plattform darstellt und gleichzeitigt immer mehr Akzeptanz in der Industrie erfährt. (siehe Abbildung 3)

## Kommunikation in einem Verteilten System

Der wohl größte Unterschied zu einem nicht verteilten System, ist die Kommunikation. Durch die zusätzlichen Wege dauert die Kommunikation deutlich länger und kommt ab einer gewissen Größe an ihre Grenzen. Sie kann außerdem immer wieder fehlschlagen und gewährleistet nie eine hundertprozentige Sicherheit. Aufgrund solcher Limitationen werden unterschiedliche Technologien benötigt und angewendet. Es können z. B. verzögerte Wiederholungen ausgeführt werden, Teilinformationen abgerufen oder gar noch gecachte Informationen weiterverwendet werden.

## Von Fehlerpotential zu Robustheit

Im Kontext von verteilten Systemen geht es nicht mehr darum ob oder falls etwas fehlschlägt, sondern wann etwas fehlschlägt. Das bedeutet, es wird fest damit gerechnet, dass Fehler auftreten werden und bereiten die Services darauf vor. Die erhöhte Kommunikation über das Netzwerk bringt dies zwangsläufig mit sich. Um dieser Erkenntnis Rechnung zu tragen, können entsprechende Komponenten und Sicherheitsmechanismen in unterschiedlichen Schichten einbauen werden, um diese Systeme robuster zu gestalten.

Einige Fehlerquellen:

1. Es können Fehler im Netzwerk auftreten.
2. Die Netzwerkbandbreite kann überschritten werden.
3. Anfragewiederholungen können Lasten noch vergrößern.
4. Nachrichten Puffer können überlaufen.
5. Fehler in einzelnen Systemen, können weitere oder gar das gesamte System zum Einsturz bringen.
6. Es können immer wieder Topologische Änderungen auftreten.
7. Es können sich Datenbank Duplikate einschleichen.

(Rotem-Gal-Oz)

Auf einige dieser Fehlerquellen soll später noch detaillierter eingegangen und die unterschiedlichen Lösungsansätze geprüft und eingeordnet werden.

## Daten Beständigkeit

Bei einem monolithischen System ist es üblich, dass es eine zentrale Datenbank gibt, wo jeglicher Datenverkehr gebündelt wird bzw. stattfindet. In einem verteilten System wird ein anderer Ansatz verwendet und jedes System verwaltet seine eigene Datenbank. Demnach findet die Kommunikation größtenteils über APIs statt und nicht über eine geteilte Datenbank.

Da die jeweiligen Systeme eigenständige Datenbanken haben können diese spezialisiert werden. Dies erlaubt verschiedene Datenbanksysteme, in einem einzigen System zu verwenden.

Auf das Thema der verteilten Datenbeständigkeit, mit all ihren Tücken und konsistenten Zuständen, wird im weiteren Teil der Arbeit nicht weiter eingegangen und sollte, bei Bedarf, zusätzlich erarbeitet werden.

## Einführung in Kubernetes

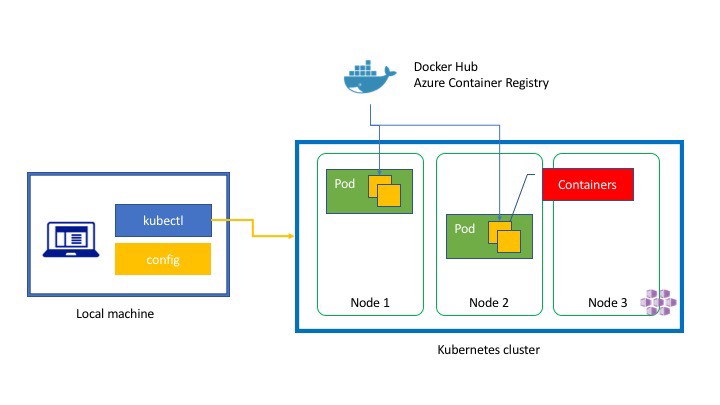
Wie es bereits zuvor unter Punkt 3.2 Orchestrierung erwähnt wurde, wird zur Durchführung dieser Thesis die Orchestrierungsplattform Kubernetes als runtime (Laufzeitumgebung) für das verteilte System verwendet. Im Folgenden werden die einzelnen Komponenten von Kubernetes erklärt, um ein grobes Verständnis darüber zu erlangen wie sie funktionieren und miteinander interagieren.

Abbildung 4: Container deployment in Kubernetes (Ushio 2018)

Um Container in Kubernetes verwenden zu können müssen sie zuerst für diese konfiguriert werden. Hierfür werden Yaml/Json Dateien benötigt, in welchen die benötigten spezifizierten Parameter festgelegt werden, wie z.B. den Namen, den Port, das zu verwendende Image usw. Durch diese Konfiguration weiß Kubernetes um welche Container Images es sich handelt und bezieht bzw. konfiguriert diese Automatisch. Für diesen Prozess wird die CLI (Command Line Interface) Kubectl verwendet, welche lokal eingerichtet ist und in einer Konfigurationsdatei, die nötigen Informationen für das momentane Cluster, gespeichert hat. Die CLI Kubectl wird für die Interaktion mit dem Cluster verwendet und ist die Schnittstelle zu diesem.

### Kubernetes Komponenten

#### Pods

Ein Pod ist immer Bestandteil eines Nodes und wird automatisch über diesen verteilt. Pods können einen oder mehrere Container beinhalten. (Abbildung 4) Jeder Container kommt mit einer Spezifikation, welche für die Repräsentation und Einstellungen in Pods benötigt wird. Falls Pods mehrere Container beinhalten, befinden sich diese im selben Netzwerk und können darüber miteinander kommunizieren, sowie auch einen möglichen Zugriff auf den gleichen Speicher vornehmen.

Mehr zu Multi-Containerkommunikation unter: <https://linchpiner.github.io/k8s-multi-container-pods.html>.

#### Nodes

Nodes zu Deutsch Knoten sind die Arbeitsmaschinen in einem Cluster, und stellen entweder eine Virtuelle Maschine, oder einen vollständigen Rechner dar. Die jeweiligen Nodes werden von der zentralen Komponente verwaltet und beinhalten alles was für den Betrieb der Pods benötigt wird. Dies umfasst die Container Runtime, das Kubelet und den Kube-Proxy.

#### Kubelet

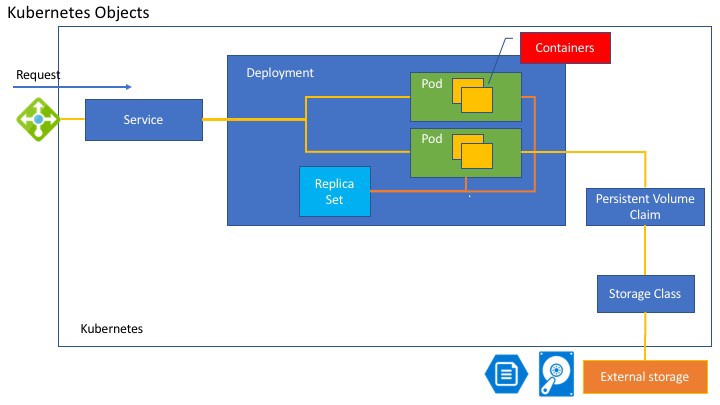
Das Kubelet ist die am niedrig stehendste Komponente in Kubernetes und ist für den Inhalt, welcher auf den einzelnen Nodes läuft, verantwortlich. Es stellt sicher, dass die jeweiligen Container laufen und bei auftretenden Abstürzen neu gestartet werden. Der API Server kommuniziert mit dem Kubelet in den jeweiligen Nodes. Das Kubelet ist dafür zuständig Informationen über die Container und deren Anwendung zu liefern und gibt diese an den API Server weiter.

#### Kube Proxy

Der Kube Proxy läuft auf jedem Node. Er spricht mit dem API Server und legt Trafficregeln in IP Tabellen fest, leitet Traffic um und filtert das Netzwerk.

#### Replica Set

Das Replica Set stellt sicher, dass immer eine festgelegte Anzahl an Pods eines Typs vorhanden und verfügbar ist (Orangene Linien in Abbildung 5). Jeder Pod wird außerdem mit einer owner Reference ausgestattet, wodurch diesen konkrete Replica Sets zugewiesen werden. Da die Replica Sets von den Deployments verwaltet und benutzt werden, kann es sein, dass diese niemals direkt manipuliert werden müssen, sondern dies durch die Spezifikation der Deployments schon erledigt wurde.

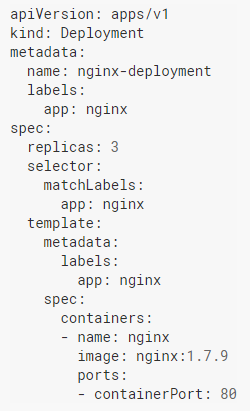
Abbildung 5: Kubernetes Komponenten und deren Interaktion (Ushio 2018)

#### Deployment

Ein Deployment bestimmt welche Pods und Replica Sets erstellt werden und beschreibt welchen erwünschten Status diese vorweisen. Der Deployment Controller führt hierbei die eigentliche Arbeit aus. Der Controller verändert das Deployment auf eine kontrollierte Weise, bis der erwünschte Istzustand erreicht ist. Deployments können später noch upgedatet, zurückgesetzt, skaliert, gestoppt und wieder fortgesetzt werden.

Da dem Erstellen von Deployments besondere Bedeutung zukommt, soll dies mit dem Blick auf ein praktisches Beispiel noch etwas verdeutlicht werden.

Deployments werden mit den schon zuvor erwähnten Yaml oder Json Dateien erstellt. Im Folgenden handelt es sich um eine Yaml Datei da diese generell der Standard ist.

*Kind:* Definiert um was es sich handelt, in diesem Fall ein Deployment.

*.metadata.name:* Definiert den Namen des Deplyoments.

*Spec.replicas:* Definiert wie viele Pods des Deployments erstellt werden sollen.

*Spec.selector.matchlabels:* Labels werden generell als Key:Value Speicher verwendet. In diesem Fall macht es dem Deployment möglich herauszufinden welche Pods zu ihm gehören.

Abbildung 6: Deplyoment Yaml (Kubernetes)

*Template.spec.\*:* Definiert welches Image, mit welcher Version, auf welchen Container Port verwendet werden soll. Images werden Standartmäßig von Docker geladen, es werden allerdings auch alle anderen gängigen Provider unterstützt.

#### Service

Services sind eine abstrakte Art, Anwendungen in Pods für das Netzwerk freizugeben. Sie dienen außerdem als Service-discovery-Mechanismus, wodurch die Anwendungen nicht verändert werden müssen. Den Pods werden eigene IP Adressen zugewiesen und DNS Namen für Gruppen an Pods verteilt, worüber anschließend eine Lastenverteilung ausgeführt werden kann.

## Vereinheitlichte Systeme

### Service-Mesh

#### Warum Service-Meshes entstanden sind

In einer Microservices Umgebungen bestehen Anwendungen für gewöhnlich aus mehreren, über das Netzwerk verbundenen, Services. Jeder dieser Services besteht aus einem oder mehreren Containern, welche im Falle von Kubernetes in Pods laufen. Wenn nun Anfragen in das System eingehen, warten diese für gewöhnlich auf Antworten von einigen Services, welche jeweils dynamisch skaliert sind. Um diese Kommunikation reibungsfrei zu ermöglichen und für Entwickler einfacher zu gestalten wurden die Service-Meshes erstellt.

Ein Service Mesh übernimmt also zum einen die Kommunikation zwischen Services und zum anderen hilft es beim Managen der Services und Komponenten. Das Service Mesh hilft also dabei Microservices zu benutzen und die Nachteile möglichst auszugleichen bzw. diese so zu handhaben, um das volle Potenzial einer Microservice Umgebung auszunutzen, in dem es das Netzwerk Management vereinfacht.

#### Was macht ein Service Mesh

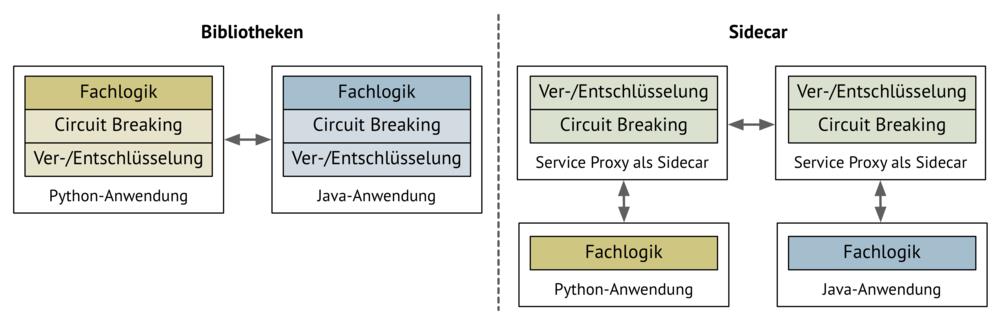
Die Microservice Architektur ist die Voraussetzung für Service Meshes, da sie genau für eine solche Umgebung konzipiert wurden und eine Infrastruktur Komponente wie Kubernetes (Orchestrierung) benutzten können. Diese wiederum stellen eine allgemeine Voraussetzung für die Service Meshes dar. Zum Bsp. das Service Mesh Linkerd 2 ist nur mit Kubernetes als Orchestrierung lauffähig. Das jeweilige Service Mesh baut auf diesen bestehenden Infrastrukturen auf und erweitert sie bzw. versucht sie einfacher benutzbar zu machen. Ein Service Mesh benutzt hierfür üblicherweise das Sidecar Pattern, welches von dem englischen Begriff Beiwagen bzw. sidecar abgeleitet ist und auch sehr ähnlich funktioniert. Die sogenannten Sidecars werden hierbei an bereits bestehende Container geheftet weshalb sie schlussendlich auch denselben Lebenszyklus besitzen. Sidecars übernehmen sprachunabhängig unterstützende Funktionen für diese Container. Typische Funktionen, welche durch Sidecars abgedeckt werden, umfassen Metriken, Routing, Ver- bzw. Entschlüsselung, Service-Discovery, Health-Checks, Robustheits-Mechanismen sowie Autorisierung und Authentisierung. Die Sidecars kommunizieren per Localhost über ein Standard-Netzwerkprotokoll mit den Containern und bilden einen weiteren Zwischen-Hop, wo ihre Funktionen angewendet werden. In Abbildung 7 ist eine Gegenüberstellung der herkömmlichen Herangehensweise, zu einer Auslagerung in Sidecars zu sehen.

Abbildung : Nutzung von Bibliotheken und die Auslagerung in Sidecars (Prinz 2019)

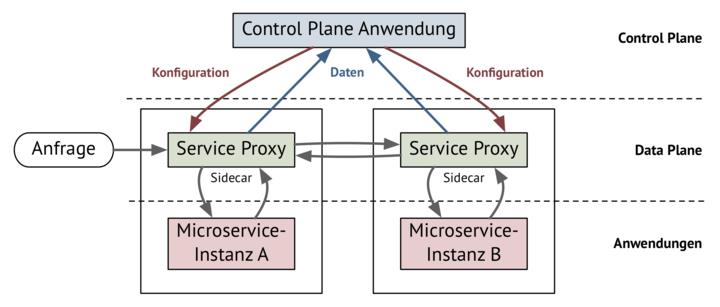
Alle Services mit ihren Sidecar Proxys werden zusammen als Data Plane bezeichnet und bilden einen von zwei großen Bestandteilen eines Service Meshes. Sidecars sind jedoch keine neue Erfindung. Was neu ist, dass sie von einem zentralen Punkt verwaltet und automatisch verteilt werden.

Abbildung 8: Service Mesh Architektur (Prinz 2019)

Den zweiten großen Bestandteil eines Service Meshes bildet die Control Plane. Die Control Plane fungiert als das zentrale Gehirn eines Service Meshes, welches mit den einzelnen Service Proxys (Sidecars) kommuniziert. Das bedeutet, die Control Plane verteilt Konfigurationen, welche zuvor in ihr eingestellt wurden, an die Proxys (siehe Abbildung 8 rote Pfeile). Weiterhin sammelt die Control Plane Monitoring Informationen der Proxys ein, (siehe Abbildung 8 blaue Pfeile) um sie an ein entsprechendes System weiterzugeben, wo sie dann verarbeitet und in z.B. in einem Prometeus Dashboard (eines von möglichen Monitoring Dashboards) wiedergeben werden kann. Im Weiteren ist eine Übersicht der Interaktion in der Architektur zu sehen.

#### Was spricht gegen die Nutzung von Service Meshes

Es gibt verschiedene Nachteile, die mit der Nutzung von Service Mesches einhergehen und hinsichtlich einer Investition in Service Meshes vorher abgewogen werden sollten. Im Moment ist ein größerer Nachteil, dass die jeweiligen Technologien relative neu sind und erst eingeschätzt werden muss wie sie sich in kleineren oder vor allem in größeren Projekten beweisen. In Anbetracht, der nicht unbedeutende Auswahl an unterschiedlichen Anbietern für Service Meshes, gestaltet sich die Übersichtlichkeit für eine aussagekräftige Einschätzung der Technologie, als sehr anspruchsvoll. Bei Bedarf kann man sich ein eigenes Bild z. B. über das Service Mesh Landscape von (Layer5 28/10/2019) machen. Zwei der bisher weiter verbreiteten Service Meshes sind Istio, jetzt in der Version 1.4 und Linkered, jetzt in der Version 2.6. Beide gelten soweit als produktionsfähig.

Ein weiterer Nachteil kommt durch die zusätzlichen Komponenten zustande. Diese können die Komplexität der Umgebung erhöhen und zudem zusätzliche Kosten verursachen, was sich wiederum belastend auf das Projektbudget auswirkt.

Außerdem erhöht sich die Latenzzeit, welche durch zusätzliche Proxys vor jedem Container entsteht und somit speziell die innere Kommunikation durch 2 weitere Hops verlangsamt wird.

Für Projekte, welche keine Microservice ähnliche Architektur benutzen, machen Service Meshes ebenso weniger Sinn, da sie ja gerade dabei helfen sollen diese zu managen. Stattdessen führt das Einführen eines Service Meshes in diesem Fall eher zu einem Overhead an Funktionalität und Ressourcen verbrauch

### Spring

#### Warum Spring ?

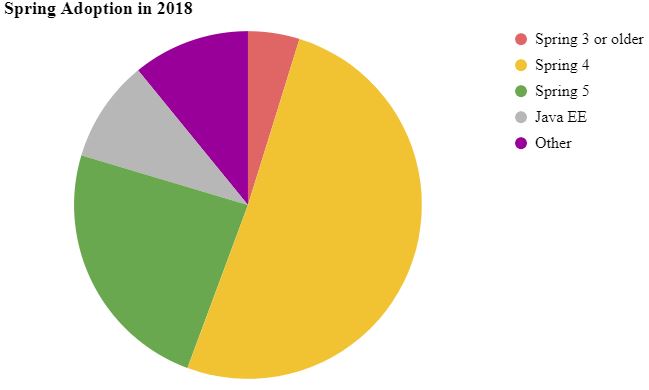
Um Microservices zu bauen wird heutzutage sehr häufig das Open-Source Framework Spring benutzt (vor allem im Java Sprachraum) „Over the last decade, Spring Framework has become the dominant framework in the Java community“. (Dashora 2019). Dieses Framework hat sich zum Ziel gesetzt die Arbeitsweise mit Java deutlich benutzerfreundlicher zu machen und einiges an Arbeit für den Entwickler zu übernehmen. Dies geschieht hauptsächlich durch eine automatische Konfiguration im Hintergrund und den exzessiven Einsatz von Annotationen. Durch diese Leistung wurde eine gute Alternative zur, bisher weit verbreiteten und zuvor präsenten Java Enterprise Edition, kurz JEE Framework, geliefert. Wie in Abbildung 9 zu sehen ist, hat das Spring Framework das ehemals vorherrschende Java EE Framework größtenteils abgelöst.

Abbildung 9: Spring Annahme (Paraschiv 2018)

Das Spring Framework wurde unter anderem vom Unternehmen Netflix mit entwickelt, welches ebenfalls eines der größten verteilten Systeme weltweit besitzt. Zur damaligen Zeit, als das Spring Framework entstand, gab es noch keine etablierten Technologien. Die vorherrschenden Technologien waren entweder nicht praktikabel oder ausreichend, um ein robustes System im erforderlichen Ausmaß zu etablieren. Dies hatte zur Folge, dass geeignete Tools entwickelt werden mussten.

Diese Tools bilden nun einen wesentlichen Bestandteil dessen, was mit einer gängigen Java Programmierung an Robustheit erzeugt werden kann. Dies beinhaltet Infrastruktur Lösungen wie Service Discovery, Distributed Tracing, verteilte Konfiguration oder Clientseitiges Load balancing sowie auch Funktionen zur Festigung der Robustheit wie Circuit Breaker, Timeouts oder auch Retry Regeln. Fast jede dieser Komponenten, kann im Java Code spezifisch konfiguriert werden und bietet dadurch deutlich mehr Möglichkeiten für Entwickler mit entsprechenden Problemen umzugehen, um z. B. Fallbacks im Circuit Breaker einzustellen. Da das Java Umfeld hierbei nie verlassen wird, fällt es Entwicklern oft deutlich leichter, Probleme direkt zu lösen, ohne ein neues System, wie ein Service Mesh zu erlernen und bietet gleichzeitig auch mehr Möglichkeiten mit Problemen umzugehen, als Konfigurationen in einem Service Mesh dies tun würden.

#### Spring Cloud

Spring Cloud ist eine Sammlung an Werkzeugen und Frameworks, welche Lösungen für gewöhnliche Muster oder Techniken in einem verteilten System bietet. Durch die Verwendung von Spring Cloud können Services und Anwendungen entwickelt werden, welche unterschiedlichste Muster oder Techniken implementieren, um in einem verteilten System koordiniert zu operieren. Diese Anwendungen ermöglichen es außerdem in jeglicher verteilten Umgebung gut zu funktionieren, wie z.B. in der lokalen Entwicklungsumgebung oder einem Datenzentrum. Die Features von Spring Cloud umschließen hierbei Konfigurationsmanagement, Service Discovery, Circuit Breaker, intelligentes Routing, micro Proxys (Sidecars), Load Balancing und einiges mehr.

### Vergleich Istio gegenüber Spring Cloud

Im Folgenden wird ein Vergleich zwischen Istio und Spring Cloud aufgezeigt. Beide werden später in der Arbeit noch anhand von Beispielen verglichen. Hier wird nun zuallererst ein mehr genereller Ansatz verfolgt, um Entscheidungen zu erleichtern und allgemeinere Informationen zu erhalten.

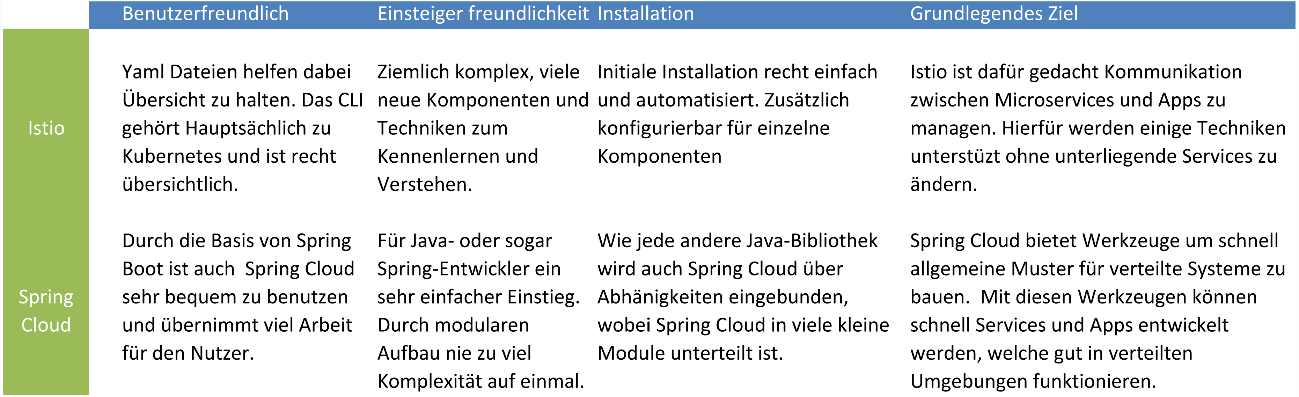


Tabelle : Spring Cloud vs. Istio 1 (Eigenkreation)

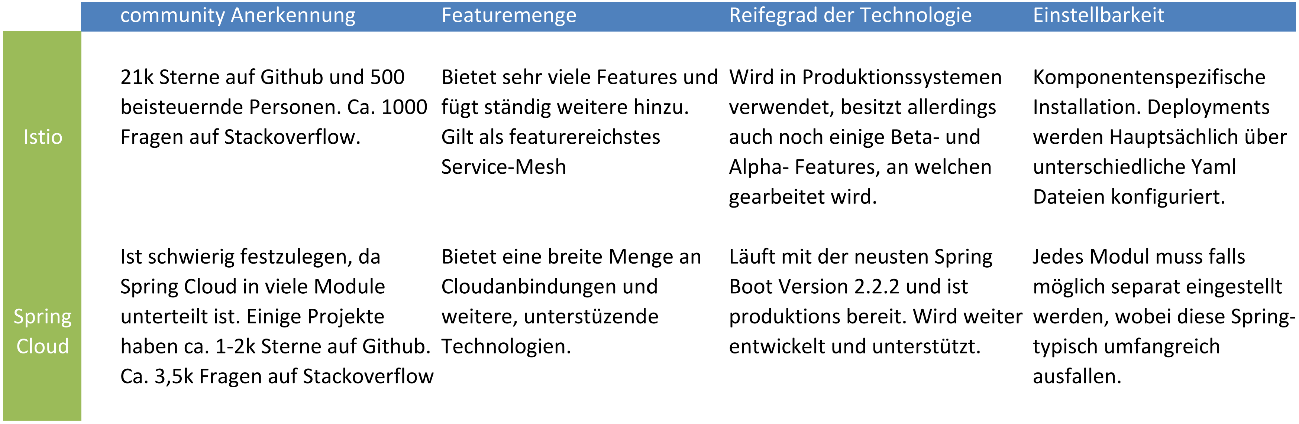


Tabelle : Spring Cloud vs. Istio 2 (Eigenkreation)

Sind noch keine Istio-Kenntnisse vorhanden, ist die Einarbeitung mit einem relativ hohen Arbeitsaufwand verbunden. Dadurch wird ein späterer Wechsel zu Istio erschwert, selbst wenn bereits Erfahrungen mit anderen Technologien vorliegen. Im Gegensatz dazu fällt eine Initiale Einarbeitung bei Spring Cloud nicht so sehr ins Gewicht, falls bereits Erfahrungen aus dem näheren Umfeld vorhanden sind.

Es sollte möglichst früh klar sein auf welcher Plattform eine Anwendung laufen soll, um nicht spezifische Komponenten zu erstellen, welche bereits auf einer der beiden Plattformen existieren. Dies ist vor allem deswegen wichtig, um zu entscheiden, ob spezifische Lösungen benötigt werden oder passende Optionen bereits vorhanden sind.

## Zusammenfassung

Zur Abrundung dieses Kapitels sollen nun noch einmal kurz die Kernpunkte festgehalten werden, da diese für das Verständnis der folgenden Kapitel als Grundlage dienen.

Es wurden die Problematiken eines verteilten Systems behandelt mit dem Schwerpunkt, dass die Kommunikation hier den entscheidenden Unterschied macht.

Der nächste Punkt betrifft die Rolle der Robustheit in einem verteilten System, um eben dieser oft alles entscheidenden Problematik vorbeugend entgegen wirken zu können. Als weiterer Punkt wurde versucht ein grobes Verständnis der Orchestrierungsplattform generell und im speziellen in Bezug auf Kubernetes zu vermitteln.

Um die im weiteren Verlauf folgenden Beispiele zu verstehen ist es äußerst relevant einen Überblick über die Service-Meshes und die Spring Umgebung zu besitzen, Grundlegendes hierzu wurde in diesem Kapitel versucht zu vermitteln.

Im Folgenden werden unterschiedliche Möglichkeiten erläutert, um die unter anderem hier beschriebenen Probleme in verteilten Systemen bzw. in einer Microservice Umgebung anzugehen und zu lösen oder den Umgang zu erleichtern. Hierbei werden einzelne Möglichkeiten herausgenommen und spezielle Beispiele gegeben.

# Möglichkeiten zum Management und Gewährleisten eines verteilten Systems

Im weiteren Verlauf werden Microservices in verteilten Systemen und die dabei auftretenden Probleme einzeln untersucht und für die jeweiligen Ansätze, falls vorhanden, unterschiedliche Lösungsmöglichkeiten vorgestellt. Es wird hierbei besonders auf die Steuerung und Kontrolle der Kommunikation eingegangen bzw. wie diese überhaupt erst ermöglicht wird. Um die anschließenden Beispiele nachvollziehen zu können ist es empfehlenswert, sich zuerst mit den theoretischen Grundlagen in den vorherigen Kapiteln auseinander gesetzt zu haben, um den Rahmen zu verstehen und ein Verständnis für die unterliegenden Systeme zu erhalten.

## Kommunikations-Leitung

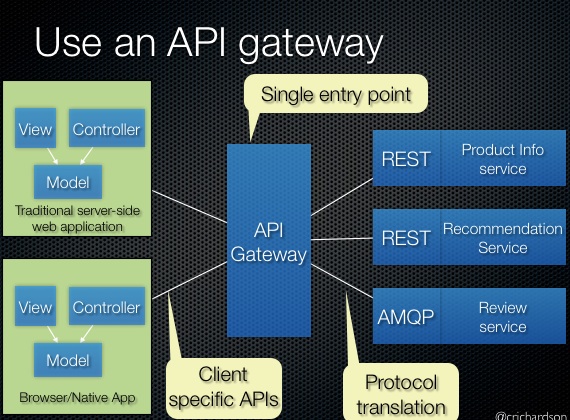
### Einleitung

In einer Microservice Architektur werden Informationen von vielen unterschiedlichen Punkten bezogen. Damit dies bei einer wachsenden Anzahl an Services weiterhin ermöglicht werden kann, werden eine oder mehrere Komponenten benötigt um diese Kommunikation möglichst automatisiert zu Steuern. Des Weiteren kommunizieren externe Nutzer mit den APIs einer Anwendung, wodurch es schwierig wird diese im Ort zu verschieben oder umzubenennen, da die Nutzer an diese fest gebunden sind. Die gesamte Kommunikation sollte Protokoll- und Sprachen-unabhängig ablaufen. Für diese Probleme gibt es unterschiedliche Lösungsansätze in der Architektur die verschiedene Anbieter unterschiedlich angehen.

### API Gateway

#### Was ein API Gateway erreichen soll

* Services können unterschiedliche, auch Web inkompatible, Protokolle nutzen siehe Abbildung 10.
* Clients benötigen Daten von vielen unterschiedlichen Quellen/Microservices, die Herkunft soll dabei vor dem Client allerdings verbogen bleiben (keine Informationen über internen Datenverkehr) siehe Abbildung 10 rechts.
* Eine client-entsprechende Datenbereitstellung sollte erreicht werden (Desktop vs. Mobile) siehe Abbildung 10 links.
* Services können sich sowohl in der Anzahl als auch im Ort bzw. der Adresse ändern und sollen vor dem Client verborgen sein, demnach keine direkte Verbindung zu ihm haben.
* Es soll auf Hardware/Netzwerk-Limitierungen, von unterschiedlichen Arten von Clients, mit entsprechenden API reagiert werden können, welche z.B. längere Wartezeiten zulassen (Abbildung 10 im Gateway).

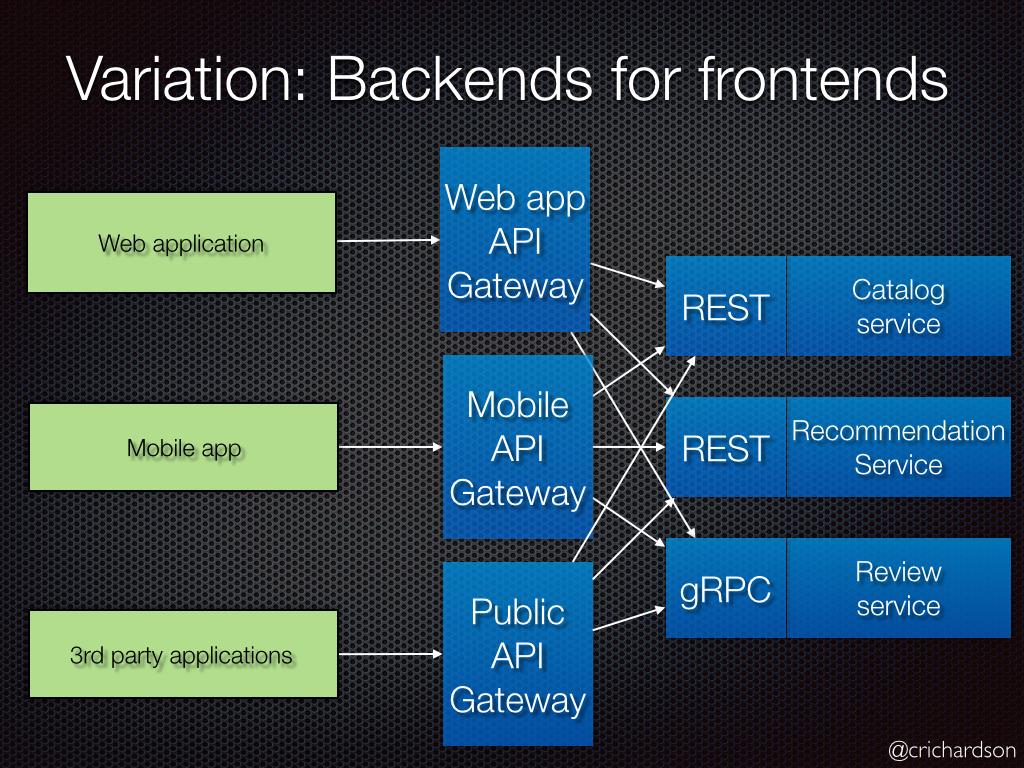
Abbildung 10: API Gateway Übersicht (Richardson 20/11/2019)

#### Ein Gateway als Lösung

Das Gateway dient als einziger Eingangspunkt für alle Nutzer. Ist dies nicht der Fall können Nachrichten unbemerkt an die APIs gelangen. Hierbei übernimmt das Gateway den gesamten Nachrichtenverkehr in beide Richtungen, dieser wird entsprechend der Einstellung des API Gateways, nur zu den entsprechenden Services durchgeschleift oder an mehrere Services ausgefächert. Das API Gateway kann jedem Nutzer eine andere API zur Verfügung stellen und dadurch die entstehenden Bedürfnisse spezifischer handhaben. Es ist auch, entsprechend abhängig von welchem Anbieter das Gateway stammt, möglich Autorisierung, Authentisieren, Logging, Monitoring usw. in dem Gateway zu verwenden und somit die Services zu entlasten bzw. Dritt-Software nicht zu benötigen.

#### Backends for Frontends

Einen etwas anderen Ansatz verfolg die Variante welche „Backends for Frontends“ genannt wird. Für jedes Frontend (sowohl Web Client als auch Mobile oder 3rd Party) wird ein eigenes Gateway erstellt, welches auch nur für dieses verantwortlich ist, siehe Abbildung 11. Das hat zum Vorteil, dass es zum einen keinen alleinigen Flaschenhals mehr gibt, zum anderen das die Gateways zweckmäßig getrennt sind und eine klarere Struktur haben. Durch ihre erhöhte modulare Gestaltung können sie außerdem besser skaliert werden und können klarer auf die Bedürfnisse ihres speziellen Clients eingehen. Zusätzliche Funktionalitäten sind natürlich immer noch möglich.

Abbildung 11: API Gateways Backends for frontends (Richardson 20/11/2019)

### API Microgateway

Das API Microgateway ist üblicherweise ein Proxy (Kommunikationsschnittstelle), entweder extern oder als Sidecar, welches vor einem Microservice sitzt. Es bietet die Möglichkeit Regeln und Sicherheitsabfragen gezielter am Service durchzuführen, den Datenverkehr direkt am Service zu überwachen, Service discovery(Erkennung) zu übernehmen und die Stabilität im Allgemeinen zu erhöhen. Solange das Gateway, micro also klein genug bleibt und nicht zu groß und schwer wird, können beliebige Funktionalitäten hinzugefügt werden. Man sollte immer bedenken, dass das Microgateway in den häufigsten Fällen im selben Bereich wie der Service läuft und infolgedessen, dessen Startzeit mit beeinträchtigt.

Jedoch wird das Microgateway durch diese enge Bauweise auch gut wiederverwendbar, was wiederum direkt zu einer besseren Skalierbarkeit führen kann. Hierbei werden nicht die API Gateways, welche am äußeren Rand des Systems sitzen, ersetzt, sondern Kommunizieren vielmehr mit diesen und erhalten Sicherheitsinformationen über z. B. API Services. Das API Microgateway wird inzwischen vor allem in Service Meshes wie zuvor beschrieben verwendet und findet dadurch eine große Anwendungsfläche.

## Kommunikations - Überwachung und - Verfolgung

### Einleitung

Bereits vor der Entstehung der Microservice Architektur wurden verteilte Systeme so groß, dass die Übersicht über diese verloren ging bzw. die Durchsuchung immer unproduktiver wurde. Dies betrifft zum einen den Überblick über die aufgerufenen Services eines einzelnen Aufrufs, zum anderen den Überblick über die unterschiedlichen APIs in einem komplexen System, welche sich in Versionierung und Typ unterscheiden. Es werden neue zentrale Punkte benötigt, welche sich mit diesen Problematiken beschäftigen und den Entwicklern einen Zugangspunkt gewähren, durch welchen sie die Möglichkeit haben, diese auf einfach Weise zu managen und den Überblick wiederherzustellen.

### API Management

#### Teilen zu unseren Bedingungen

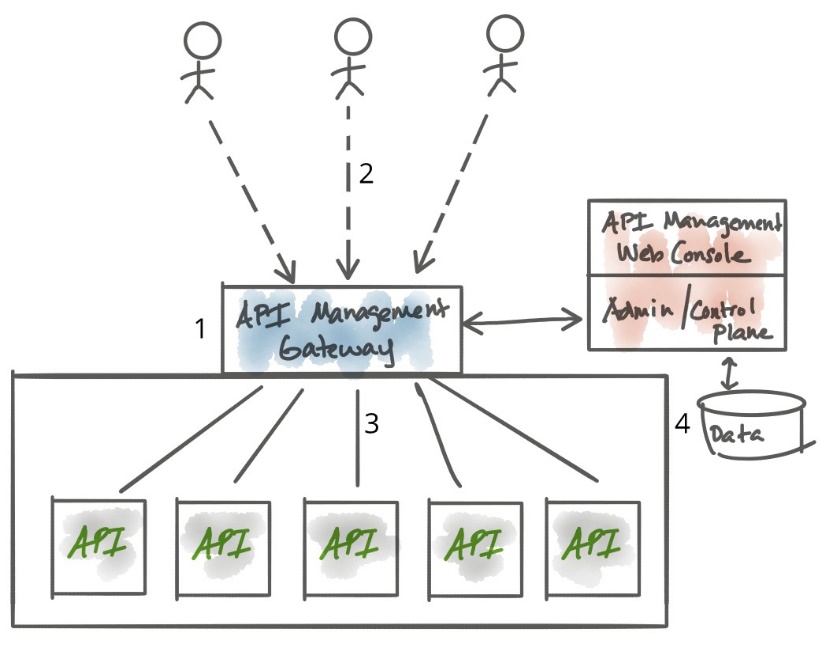
Einen der zuvor erwähnten zentralen Punkte bildet das API Management Gateway. Dieses Management Gateway deckt, entsprechend dem Anbieter, gewisse Funktionalitäten ab und soll das Leben von Entwicklern einfacher machen, indem es z.B. APIs verwalten lässt.

“We want to solve the problem of we have these existing, curated, APIs that we want to share with others but share them on our terms”. (Posta 2019)

#### Das API Management Gateway

Das API Management Gateway ist eine Komponente, welche eine weitere Schicht bildet oder das bisherige Gateway komplett ersetzt (dies kann auch von der Größe des Systems abhängen). Es managt wann existierende APIs für Konsumer erreichbar/benutzbar sind und notiert dessen Nutzung, etabliert Regeln und hält fest für wen sie gelten, der Sicherheitsfluss wird festgelegt und ergibt Freigaben für die Nutzung (Autorisierung, Authentisierung). Alle APIs, Abbildung 12 [3], werden von diesem Gateway gemanagt, katalogisiert und verwaltet, wodurch diese vergeben, gefunden und effektiv kontrolliert werden können.

Die Managementkomponente kann als Einstiegspunkt in das System dienen, wie es in Abbildung 12 [2] zu sehen ist, muss dies allerdings nicht. Es kann so auch ein dezentralisierter Ansatz verwendet werden, wodurch Gateways direkt vor jede API gesetzt werden und somit das System besser skalierbar ist.

Abbildung 12: API Management Gateway Overview (Posta 2019)

Dabei muss beachtet werden, dass keine Businesslogik in diese Schicht, Abbildung 12 [1] mit einfließt. Da das API Management eine geteilte Komponente ist hat es Tendenzen ein allwissendes, allverarbeitendes Konstrukt zu werden, wo jeglicher Datenverkehr für Änderungen an APIs durchfließen muss, was erneut in einem organisatorischen Engpass enden kann. Zumal auch die Managementkomponente skaliert werden muss sollte hier möglichst versucht werden keine Abhängigkeiten zu anderen Komponenten zu erzeugen weshalb auch eine eigene lokale Datenbank zur Verfügung gestellt werden sollte, Abbildung 12 [4].

#### API Management VS Service Mesh

Nachdem nun ein grundlegendes Verständnis über API-Management und Service Meshes (Kapitel 3.7.1) besteht, wird nachfolgend der Unterschied dieser beiden Systeme und wie sie sich gegenseitig komplementieren können, untersucht.

In einem API Management steht ein Gateway in der Regel vor einer Gruppe an Microservices, welche eine Anwendung ausmachen. Aufrufer dieser Microservices/APIs benötigen kein Wissen darüber wie diese APIs funktionieren bzw. implementiert sind. Für die Aufrufer ist es vollkommen unwichtig wie die Services skaliert oder repliziert werden. Das Einzige was die Benutzer interessiert, ist wie sie zum Gateway kommen, welches sie dann zu jeglichem angefragten Endpunkt weiterleitet.

Im Gegensatz dazu, benutzt ein Service Mesh typischerweise das Sidecar-Muster, welches das Gateway überflüssig macht. In diesem Fall befinden sich die Aufrufer innerhalb des Meshes und müssen sich dessen auch bewusst sein. Ein weiterer Unterschied besteht darin, dass clientseitiger Code zum Service Mesh dazu gehört und z.B. weiß wie der Discovery-Mechanismus und das Load-Balancing funktioniert. Hierdurch muss nicht länger Code in der aufrufenden Komponente vorhanden sein, sondern wird automatisch vom Sidecar implementiert, welcher neben den Services läuft. Die Sidecars sind, logisch gesehen, ein Teil des Clients, da die meisten zwischenkommunikativen Tätigkeiten von ihm durchgeführt werden. Das unterscheidet ihn grundsätzlich vom API Management, welches die meisten Tätigkeiten serverseitig durchführt, wie in der folgenden Tabelle (Tabelle 3) ersichtlich ist.



Tabelle 3: Service Mesh VS API Management (Eigenkreation)

Service Meshes wie Istio mit einer API Management Lösung sind noch nicht allzu verbreitet. Ein Beispiel hierfür wäre die Open Source Lösung von wso2:

https://wso2.com/api-management/microservices/istio/

In diesem System sitzt das API Management über dem Service Mesh Istio und hilft dabei diese zu verwenden und die Services zu veröffentlichen. Durch diese Verknüpfung ist es möglich einem Kunden schneller eine Lösung zu präsentieren und APIs zu veröffentlichen. Außerdem erhält man weitere Daten, welche zuvor so nicht vorhanden waren, wie die gesamte Anzahl an Aufrufen der APIs.

### Distributed tracing

#### Was ist Distributed tracing

Bei Distributed Tracing geht es darum Anfragen, über mehrere Systeme hinweg, zu verknüpfen. Die Verknüpfung geschieht durch zugehörige, eindeutige IDs, welche im http-Header mitgeliefert werden. Die IDs werden in Zusammenhang gebracht, um so den gesamten Weg einer Anfrage nachvollziehen zu können. Das bedeutet, es ist einfacher möglich herauszufinden wo ein Fehler tatsächlich entstanden ist und welche Systemkomponente verantwortlich ist.

#### Wieso Distributed tracing wichtig ist

Mit der Architekturänderung von Monolithen zu Microservices haben sich einige neue Herausforderungen ergeben. So sind wenige lokale Module in viele verteilte Services umstrukturiert worden und was „zuvor“ noch lokal debugt werden konnte, verteilt sich nun über das Netzwerk hinweg über etliche Hops.



Abbildung 13: Distributed Tracing (Kyma)

Da niemand gerne viel Zeit damit verbringt sich durch etliche Netzwerklogs durchzuwühlen wird eine automatisierte Lösung benötigt. Um dieses Problem in den Griff zu bekommen benötigt es neue Analysewerkzeuge und hier kommt Distributed Tracing ins Spiel.

#### Nachteile

Wie so vieles, hat auch Distributed Tracing einige Nachteile. Die anfallenden Logs können immense Dimensionen erreichen und sind fast nicht komprimierbar, was vor allem an ihrer notwendigen Einzigartigkeit liegt. Sie können je nach Systemgröße sehr groß/viele werden „Tracing is high-volume and high-cardinality“ (McAllister 2019, 6:5-6:10). Es gibt einige verschiedene „Standards“ (anstatt einen) was dazu führen kann, dass ein Anbieterwechsel signifikant erschwert wird, "When you follow widely adopted standards you get to avoid vendor lockin which is actually pretty important inside of this space" (McAllister 2019, 15:15-15:25). Eine Codeanpassung für Distributed Tracing kann sehr umfangreich sein, sodass es niemand zweimal durchführen möchte. "Trust me instrumenting all of your code once is more than enough" (McAllister 2019, 15:37-15:42).

## Gewährleistung von Services

### Einleitung

Mit einer steigenden Anzahl an Microservices und der damit erhöhten Kommunikation entstehen unweigerlich neue Probleme. Um diesen vorbeugend entgegenzuwirken und vollständig einsatzbereite robuste Systeme zu erreichen, werden neue Techniken benötigt, welche im Folgenden genauer betrachtet und die damit verbunden Probleme beleuchtet werden.

### Resilient4j VS Failsafe

Für die Implementierung in Java wird dabei auf Resilient4j gesetzt und im Weiteren, zweierlei Alternativen, mit deren Vor- und Nachteilen, aufgezeigt.

Resilient4j ist hierbei jene neue Bibliothek, welches in Zukunft die vermutlich größte Akzeptanz erfahren und somit denn bisherigen Platzhirsch Hystrix von Netzflix ablösen wird, da dieses nicht länger aktiv weiterentwickelt wird.

Eine Alternative zu Resilient4j stellt Failsafe dar.

* Failsafe hat hierbei keine weiteren Abhängigkeiten welche später zu Problemen führen können (unterschiedliche Versionen oder Überschneidungen mit eigenen Abhängigkeiten), wohingegen Resilient4j eine Abhängigkeit auf Vavr besitzt.
* Failsafe bietet die Möglichkeit über ein SPI (Service Provider Interface) zusätzliche Regeln zu implementieren.
* Failsafe lässt asynchrone Aufrufe jederzeit abbrechen und unterbrechen.
* Failsafe unterstützt asynchrone API Einbindungen, wodurch die Vollendung von fremdem Code immer noch durch Callbacks bestätigt wird.
* Beide Bibliotheken bezeichnen sich als leichtgewichtig.
* Beide kommen bei einigen Unternehmen zur Anwendung, wobei Failsafe hier deutlich stärker referenziert ist.
* Beide Bibliotheken sind Open Source, allerdings sind an Resilient4j ca. 5-8 Personen mit größerem Anteil beteiligt, wohingegen bei Failsafe dies allerhöchstens 2 Personen sind und von einer Person getragen wird.
* Resilient4j hat 5k Sterne auf Github wohingegen Failsafe 3,1k Sterne besitzt.
* Resilient4j bietet eine ausführlichere Dokumentation.
* Resilient4j vereinfacht die Nutzung anderer Bibliotheken wie Spring Boot.
* Resilient4j stellt Einstellungen über Konfigurationsdateien ein und unterscheidet diese im Namen.
* Resilient4j bietet zusätzliche Robustheitsmöglichkeiten, wie das Bulkhead und das Rate Limiting.

#### Einschätzung

Diese beiden Bibliotheken unterscheiden sich vor allem in ihrem Ansatz. Resilient4j bietet eine breitere Reihe an Möglichkeiten und macht diese einfacher benutzbar. Failsafe auf der anderen Seite wählt einen Ansatz, welcher mehr Eigenentwicklung benötigt, aber auch mehr Freiraum gibt, wobei asynchrone Programmierung besonders stark Fokussiert wird.

### Bulkhead

#### Was sind Bulkheads

Bulkhead ist ein Begriff, welcher aus dem Schiffsbau kommt und eine Technik, bei der das Schiff in Segmente unterteilt wird, beschreibt. Bei einem Leck können diese Segmente separat geschlossen werden und somit verhindern, dass das gesamte Schiff voll Wasser läuft (siehe Abbildung 14 und Abbildung 15). Feuerschutztüren im Gebäudebrandschutz dienen demselben Zweck und verhindern die Rauchverbreitung im Haus.

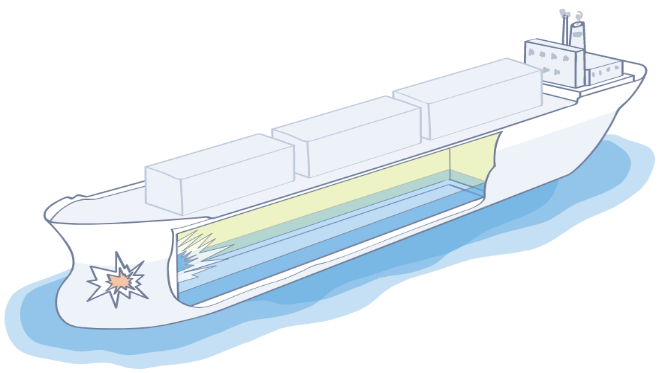
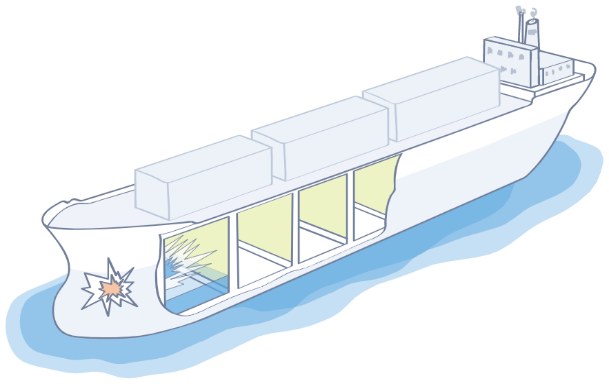
Abbildung 14: Schiff Links Ohne Bulkhead (IBM 28/11/2019)

Abbildung 15: Schiff Rechts mit Bulkhead (IBM 28/11/2019)

Sowie im Schiffsbau und im Brandschutz, wird das Bulkhead Pattern (in Deutsch Schott Muster) auch in der Microservice Architektur verwendet, um zu verhindern, dass das gesamte System durch einen einzigen Fehler zum Absturz gebracht wird. Dies geschieht im Falle von Microservices auf die Art und Weise, dass einzelne Bahnen, im Prozessablauf von Beginn an voneinander technisch getrennt/isoliert werden. Das bedeutet, dass ein Service nicht auf dieselbe Warteschlange oder Threadpool zugreift wie ein anderer Service, sondern diese separat zugewiesen wird.

Einerseits wird durch dieses Pattern der Schaden zwar nicht verhindert, allerdings wird er davon abgehalten sich auszubreiten, wodurch alle anderen Services weiterlaufen können. Andererseits dient es dazu nur jeweils so viel Arbeit auf einmal abzuarbeiten wie eingestellt ist, um immer reagieren zu können. Wenn man beim Bild des Schiffes bleibt, wiederspiegeln die Kammern jeweils einen Service. Jede Kammer hat eine gewisse Anzahl an Ressourcen und jeder Service kann nur so viel Einnehmen/Fluten wie seine Kammer hergibt/bereitstellt.

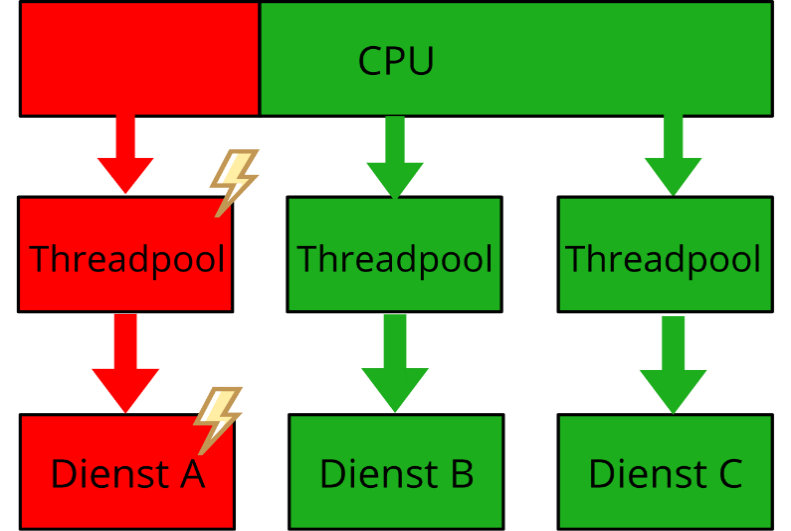
#### Microservices Probleme ohne Bulkheads

Abbildung 16: Fehlerhafter Service ohne Bulkheads (Eigenkreation)

Wenn in einem Softwaresystem ein Fehler verursacht wird stürzt in der Regel das gesamte System ab. In einem verteilten System bedeutet das, sollte ein Fehler in einem Teilsystem verursacht werden, dass alle verbundenen Services oder gar das gesamte System abstürzt. Am Beispiel der Abbildung 16 verursacht der Dienst-A Fehler, wodurch dieser Anfragen sofort beantwortet und erneut Anfragen erhält. Durch diesen Prozess werden jegliche Ressourcen des Threadpools und damit der CPU vereinnahmt wodurch andere Dienste keine Möglichkeit mehr haben selbst zu operieren.

#### Microservices mit Bulkheads

Wenn Services mit der Bulkhead Technik ausgestattet sind sieht das Verhalten mehr wie in Abbildung 17 zu sehen ist aus. Hier wurde jeder Service mit einem eigenem Threadpool ausgestattet, wodurch zwar einer dieser Pools ausgeschöpft und damit entsprechend viel Last auch auf die CPU übertragen wurde, die Restlichen Services jedoch zur selben Zeit ohne Beeinträchtigung weiterlaufen können.

Abbildung 17: Fehlerhafter Service mit Bulkhead (Eigenkreation)

#### Prinzipien des Bulkhead Patterns

1. Teile möglichst nichts:

Soweit es möglich ist sollte, wenn Services in eigene Fehlerzonen isoliert werden, keine Datenbanken, Firewalls, Speicher und Rechenleistung etc. geteilt werden. Aus Kostengründen kann es auch nur auf die Services heruntergebrochen werden.

1. Vermeide synchrone Aufrufe zu anderen Services:

Synchrone Service-zu-Service-Kommunikation erweitert die Fehlerzone eines Bulkheads. Es können Fehler und Trägheit mit synchronen Aufrufen übertragen und somit den Schutz, welcher durch ein Bulkhead gewährleistet wird, verletzt werden.

#### Umsetzung

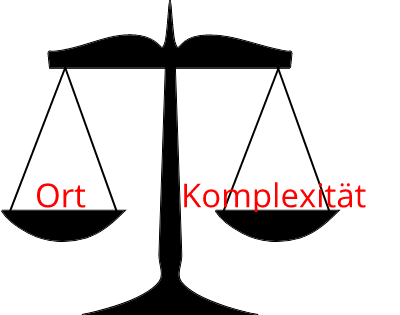
Es gibt verschiedene Möglichkeiten einen Bulkhead in ein Softwaresystem einzubauen. Dies unterscheidet sich einmal im Ort der Implementierung und zum anderen in der Komplexität der Implementierung. Verschiedene **Orte** wären in diesem Falle eine direkte Implementierung im Code, eine Nutzung von Drittanbieter Software, eine Einschränkung durch Firewallregeln oder durch Konfiguration in einem Service Mesh.

Abbildung 18: Implementierungswaage (Eigenkreation)

Die **Komplexität** der Implementierung sagt aus wie spezifisch auf das Geschehen des Prozesses Einfluss genommen werden kann. Diese beiden Eigenschaften sind oft wie zwei Seiten einer Waage (Abbildung 18). Falls es bevorzugt wird, eine Implementierung direkt im Code durchzuführen, können automatisch auch spezifischere Lösungen produziert werden. Ist ein Ansatz in einem höheren Level gewünscht, wodurch auch weniger Code selbst geschrieben werden muss, so kann oft auf bereits bestehende Funktionen zurückgegriffen werden. Durch die Verwendung von Funktionen in höheren Ebenen wird die Umsetzung allerdings auch automatisch limitiert, weshalb nur teilweise Funktionen wie Rate-Limiting eingebunden werden können. Da kein direkter Zugriff auf prozessinterne Ressourcen besteht stellt sich hierbei auch die Frage, kann überhaupt von einem Bulkhead die Rede sein, wenn eine Anwendung nicht in unterschiedliche Ressourcen-abschnitte aufgeteilt werden kann.

##### Beispiel: Eine Code Umsetzung in Java mit dem Resilient4j.

Wie im Code Beispiel, Codeteil 1, im ersten Abschnitt, zu sehen ist, sind in einem in Java implementierten Bulkhead sehr spezifische Änderungen möglich, wodurch zum einen dieser Service in seinen eigenen Threadpool eingeschlossen wird und zum anderen dieser Threadpool durch spezifische Einstellungen eingeschränkt werden kann. Durch diese direkte Implementierung können solche spezifischen Eigenschaften für jeweilige Routen/APIs festgelegt werden, ohne die bekannte Entwicklungsumgebung bzw. Sprache zu verlassen. Darüber hinaus hilft Resilient4j dem Entwickler noch dabei effizienteren Code zu schreiben, indem es falls ein Thread Pool Bulkhead verwendet wird, asynchrone Kommunikation erzwingt, welche in einer Microservice Umgebung bereits Standard sein sollte. Falls die synchrone, semaphore Version des Bulkheads verwendet wird, fallen allerdings einige Einstellungsmöglichkeiten weg, weshalb kein allzu genauer Einfluss auf den Bulkhead genommen werden kann.

ThreadPoolBulkheadConfig threadPoolBulkheadConfig = ThreadPoolBulkheadConfig.*custom*()  
 .maxThreadPoolSize(4)  
 .coreThreadPoolSize(2)  
 .queueCapacity(8)  
 .keepAliveDuration(Duration.*ofMillis*(100))  
 .build();  
  
ThreadPoolBulkheadRegistry threadPoolBulkheadRegistry = ThreadPoolBulkheadRegistry.*of*(threadPoolBulkheadConfig);  
**threadBulkhead** = threadPoolBulkheadRegistry.bulkhead(**"threadBulkhead"**);  
  
Runnable runnable = ()-> **todoRepository**.findAll().forEach(todo -> todos.add(todo.getTodo()));  
**­­­­threadBulkhead**.executeRunnable(runnable);

Codeteil 1: Bulkhead Serverseitige Implementierung in Java mit resilience4j in Plain Java

io.github.resilience4j.bulkhead.BulkheadFullException: Bulkhead 'threadBulkhead' is full and does not permit further calls

Codeteil 2: Bulkhead Fehlermeldung wenn der Threadpool vollgelaufen ist.

@GetMapping(path = **"resilience"**)  
@Bulkhead(name=**"BACKEND"**,type = Bulkhead.Type.***THREADPOOL***)  
**public** CompletableFuture<ModelAndView> getItems()  
{  
 String MethodName = **"Standard"**;  
 **return** CompletableFuture.*completedFuture*(getTodos(MethodName));  
}

Codeteil 3: Resilient4j mit Spring Boot 2.

Durch die Verknüpfung von Resilient4j mit Spring Boot 2 wird die Verwendung der Resilient-Mechanismen um einiges einfacher wie zuvor im reinen Javacode (siehe Codeteil 3). Die alleinige Voraussetzung zur Nutzung des Threadpool Bulkheads anstelle des Semaphoren Bulkheads ist die, dass ein CompletableFuture Objekt benutzt wird. Dies ist sehr einfach realisierbar, weil das Objekt von der Methode zurückgegeben werden kann. Das CompletabelFuture Objekt wird in Java für asynchrone Verarbeitung verwendet und erleichtert dies seit Java 8 mit einer reichhaltigen API. Die Einstellungen für das Bulkhead, werden in einer application.properties oder .yml Datei festgelegt und bieten viel Spielraum zum Einstellen der Funktionalität. Falls mehrere Resilient- Funktionalitäten verbindet werden möchten, geschieht dies von Haus aus in folgender Reihenfolge:

Retry ( CircuitBreaker ( RateLimiter ( Bulkhead ( Function ) ) ) ).

##### Bulkhead in Kubernetes/Istio

Beide Fälle befinden sich in der Netzwerk-Ebene, wodurch ein Eingreifen in Prozesse einer Anwendung praktisch unmöglich wird. Dementsprechend handelt es sich, wenn immer von Bulkheads im Zusammenhang mit einem der Beiden gesprochen wird, nicht um Bulkheads, sondern viel eher um ein Rate-Limiting.

Auf Rate Limiting wiederum wird später im Kapitel 4.3.6 Rate Limiting genauer eingegangen.

### Circuit Breaker

#### Ausfallursachen in Softwaresystemen

Ein Ausfall kann sich entwickeln, wenn zu viel Last/Strom auf einer elektrischen Leitung liegt und diese infolgedessen durchbrennen würde, was in Folge zu einem Feuer führen kann. In einem Softwaresystem sieht das etwas anders aus. Ausfälle geschehen hier, wenn so viel Last an einem Service anliegt, dass dieser zum Absturz gebracht wird und dieser infolgedessen versucht neu zu starten. Während der Startphase des Service wirkt nun weiterhin so viel Last auf den Service ein, dass dieser sofort wieder abstürzt und sich nicht wieder erholen kann, wodurch ein dauerhafter Ausfall oder nicht erreichbarer Service entsteht.

#### Was sind Circuit Breaker

Circuit Breaker, zu deutsch Sicherung, kommen ursprünglich aus dem elektrischen Bereich. Sicherungen sind kleine Drähte oder Widerstände, welche bei einer vorgegebenen Überlast durchbrennen, bevor Hauptleitungen z. B. in der Wand durch übermäßige Erwärmung ein Feuer auslösen können. Dies hat ursprünglich öfters zu Hausbränden geführt. Genauso werden sie als Notabschaltung benutzt, um größere Schäden zu verhindern.

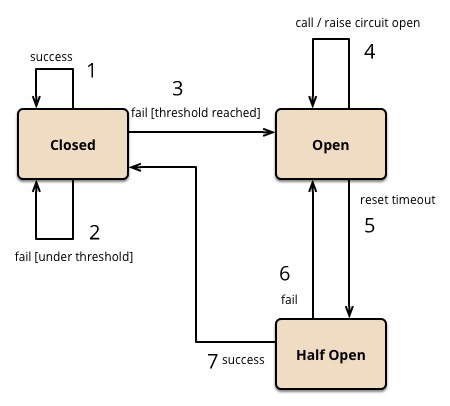
Dieses Prinzip, wurde nun anhand eines Programmiermusters in der Software übernommen, umso eine erhöhte Stabilität zu gewehrleisten. Wie in der Elektrotechnik gibt es auch hier mehrere Zustände eines „Schaltkreises“. Er kann offen, geschlossen oder zusätzlich halb-offen sein. Die Einzelnen Zustände werden nun anhand des Folgenden Bildes (siehe Abbildung 19) erklärt.

Abbildung 19: Circuit Breaker Zustände (Fowler 2014)

**Closed** ist zuallererst der Zustand, welcher den normalen Betrieb widerspiegelt [1], dieser wird nur verlassen, wenn eine entsprechende Anzahl an Fehlern in einer festgelegten Zeit überschritten wird ([2] threshold). Verlassen bedeutet, in den Open Status zu wechseln [3]. Hierbei ist es wichtig zu unterscheiden um welche Arten von Fehlern es sich dabei handelt. So kann es sich z.B. nur um Verbindungsfehler, um Zeitüberschreitungen oder andere Fehler handeln.

**Open**, blockt erst einmal jeglichen Datenverkehr [4] und schaltet erst nach einer gewissen Zeit auf Half-Open um [5]. Im Status OPEN werden sofort Fehlermeldungen an die anfordernden Aufrufer abgesetzt. In dieser Zeit soll sich der Service, von dem aufkommenden Datenverkehr erholen können oder Zeit haben neu zu starten. Die Wiederherstellung der Verbindung wird in der Regel von externen Systemen übernommen und findet nicht im Circuit Breaker statt.

**Half-Open** testet den Service. Es werden hierbei eine kleine Anzahl an Anfragen angenommen die normalerweise geringer ist als im ursprünglichen Closed Status. Wenn dieser Test erfolgreich verlaufen ist, wird wieder in den Closed Zustand gewechselt [7]. Falls die Anfragen weiterhin fehlschlagen wird wieder zurück in den Open Status gewechselt [6].

#### Unterschiedliche Arten/Ebenen von Circuit Breakern

Man kann im Allgemeinen zwei unterschiedliche Arten von Circuit Breakern unterscheiden. Unterschieden wird nach erforderlichem Managementaufwand und möglicher Einflussname, was Hand in Hand mit erhöhtem Programmieraufwand geht.

#### Zentral Verwaltete Circuit Breaker

Diese Art eines Circuit Breakers beschreibt einen von einer Middleware verwalteten Circuit Breaker. Normalerweise ist dies ein API Gateway, ein Service Mesh oder ein Reverse Proxy. In diesen Fällen läuft der gesamte Datenverkehr über diese Middleware und wird von dort weitergeleitet. Das größte Problem ist die Gefahr, dass dies zu einer einzelnen Schwachstelle für das gesamte System werden kann. Die Infrastruktur Architekten sollten hier immer besonders darauf achten, dass diese Schwachstellen möglichst ausfallsicher betrieben werden. Auf der anderen Seite hilft es den Entwicklern, da sie sich weniger mit diesem zusätzlichen Konstrukt beschäftigen bzw. dieses entwickeln müssen.

#### Unabhängig Verwaltete Circuit Breaker

Ein unabhängig verwalteter Circuit Breaker steht für einen im Service selbst implementierten Circuit Breaker welcher den ankommenden Datenverkehr regelt. Ein großer Vorteil hierbei ist, dass auf diese Art und Weise keine einzelnen großen Schwachstellen mehr vorhanden sind, da diese nach unten auf die Services verschoben werden, was speziell in hoch verfügbaren Systemen wichtig wird. Der Nachteil ist die erhöhte Komplexität, welche für die Entwickler hinzukommt. Das ist vor allem der Fall, wenn unterschiedliche Service in verschiedenen Programmiersprachen geschrieben werden oder Entwickler noch unerfahren auf dem Gebiet der weiterführenden Fehler sind. Es gibt allerdings in allerlei Sprachen gut geschriebene Bibliotheken, welche weit verbreitet sind und benutzt werden können, was dabei hilft, diese Probleme in den Griff zu bekommen.

#### Umsetzung

Da Circuit Breaker mit Nachrichten arbeiten, können diese auf fast allen gängigen Ebenen eingesetzt werden. Sie unterscheiden sich bezüglich des Ortes an dem sie ausgeführt werden, wie spezifisch diese eingestellt werden können und wie einfach die Anwendung ist. Es kann von Vorteil sein einen Circuit Breaker möglichst früh im eigenen Netzwerk einzubauen, um darunterliegende Systeme zu schützen.

##### Eine Code Umsetzung in Java mit Resilient4j und Spring Boot 2.

Um Resilient4j mit Spring Boot 2 benutzen zu können müssen lediglich einige Abhängigkeiten erfüllt werden, sowie die entsprechenden Methoden mit Circuit Breaker Annotation und entsprechenden Namen versehen werden (siehe Codeteil 4). Die jeweiligen Namen sowie Einstellungen, für jegliche Eigenschaften werden in einer application.properties oder application.yml Datei festgelegt, wie im Codeteil 5 zu sehen ist, wobei BACKEND den Namen des Circuit Breakers widerspiegelt.

@GetMapping(path = **"cb"**)  
@CircuitBreaker(name=**"BACKEND"**)  
**public** ModelAndView getItems()  
{

Codeteil 4: Methode welche mit Resilient Annotationen des Circuit Breakers versehen ist.

**resilience4j.circuitbreaker.instances.BACKEND.wait-duration-in-open-state**=**500  
resilience4j.circuitbreaker.instances.BACKEND.permitted-number-of-calls-in-half-open-state**=**2**

Codeteil 5: Circuit Breaker Einstellungen in der application.properties

##### Circuit Breaker in Istio

Für die Anwendung in Istio wird davon ausgegangen, dass die Pods automatisch mit Sidecars injiziert werden und keine gegenseitige Authentifizierung (mutual TLS) aktiviert ist.

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

  name: myService

spec:

  host: myService

  trafficPolicy:

    connectionPool:

      tcp:

        maxConnections: 1

      http:

        http1MaxPendingRequests: 1

        maxRequestsPerConnection: 1

    outlierDetection:

      consecutiveErrors: 1

      interval: 1s

      baseEjectionTime: 3m

      maxEjectionPercent: 100

Codeteil 6: Cicuit Breaker durch eine destination Rule in Istio

Im Codeteil 6 sieht man eine yaml Datei, welche eine Destination Rule für einen Service definiert. Destination rules, wenden Regeln auf den Netzwerkverkehr von Services an nachdem das eigentliche Routing geschehen ist. Der Servicename ist in diesem Fall myService und läuft unter den zuvor beschriebenen Parametern. Der Circuit Breaker befindet sich unter dem Outlier Detection Feld, worunter dieser auch eingestellt wird. Der Connection Pool Parameter und seine Kinder definieren hierbei nur wie viele Verbindungen zum Service aufgebaut werden dürfen. In Istio ist kein Wort wörtlicher Circuit Breaker vorhanden, stattdessen wird dies durch Netzwerkregeln gelöst, welche z.B. in Spring in einer application.properties Datei festgelegt werden. Circuit Breaker werden in Istio außerdem direkt für einen ganzen Services festgelegt und können nicht für eine spezifische Route definiert oder spezifiziert werden. Da die Regeln sich stärker auf das Netzwerk beziehen sind im Falle von Istio nicht ganz so viele Einstellungsmöglichkeiten vorhanden, was die Bedienung allerdings vereinfachen kann. Die oben zu sehenden Regeln müssen nur per Kommando: *kubectl apply -f .\destinationRule.yaml* angewendet werden, um anschließend für den entsprechenden Service in Kraft zu treten.

### Retry

#### Das zu Lösende Problem

Wenn Anwendungen über das Netzwerk miteinander kommunizieren, kann es immer wieder vorkommen, dass gewisse Fehler auftreten. Beispielsweise können Anfragen verloren gehen, das Ziel kann momentan nicht erreichbar sein oder ein Service vorübergehend ausgelastet sein. Solche Fehler verschwinden mit der Zeit, für gewöhnlich, von alleine. Wir brauchen allerdings einen Mechanismus, der dies möglichst einfach umsetzbar macht, ohne zu große weitere Probleme zu verursachen.

#### Lösungsansatz für das Problems

Es kann nun also vermerkt werden, dass Fehler immer wieder auftreten, welche gehandhabt werden müssen. Stellt die Anwendung also fest, dass ein Fehler aufgetreten ist, beim Versuch einen Aufruf auszuführen besteht eine Reihe an Möglichkeiten damit zu verfahren:

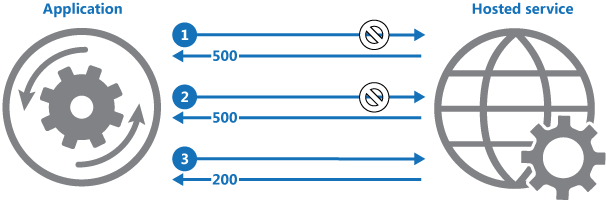
* Abbruch: Falls erkennbar ist, dass der Fehler sich nicht mit einer Wartezeit beheben lässt oder mit dem Kommunikationsweg zu tun hat, sollte die Aktion abgebrochen und eine Exception ausgelöst werden. Wird beispielsweise ein Login mehrfach mit den falschen Daten ausgeführt, macht es keinen Sinn diesen überhaupt abzusenden.
* Retry: Kommt ein Fehler nur selten vor und weist keine Rückschlüsse auf konkrete Fehler auf, sollte dieser einfach sofort wiederholt werden, da hier mit erhöhter Wahrscheinlichkeit das Netzwerk die tatsächliche Fehlerquelle darstellt, was bedeutet, dass der Fehler nicht nochmal auftreten wird (Bsp. Abbildung 20).

Abbildung 20: Retry Pattern (Microsoft)

* Retry mit Verzögerung: Falls es sich um einen eher gewöhnlicheren Fehler handelt, z. B. der angefragte Service oder das Netzwerk sind überlaufen, ist die beste Option die Anfrage mit einer gewissen Verzögerung erneut zu senden.

#### Umsetzung

##### Retries in Spring Boot 2 mit Resilient4j

Die Retry Funktion von Resilient4j in Verbindung mit Spring Boot 2 ist sehr einfach einzubinden und bietet auch einiges an zusätzlichen Einstellungen, welche bei anderen Alternativen so nicht vorhanden sind. Hierfür wird jede Klasse oder Methode, wie in Codeteil 8 zu sehen ist, zum einen mit der entsprechenden Annotation erweitert, sowie mit einem Namen, und falls gewollt einer Fallback Methode versehen. Wie im Codeteil 7 zu sehen ist, wird das Verhalten des Retry Mechanismus in der application.propteries Datei dem Namen entsprechend festgelegt.

resilience4j.retry.instances.BACKEND.max-retry-attempts=2  
resilience4j.retry.instances.BACKEND.wait-duration=10ms

Codeteil 7: Retries Einstellungen

@Retry(name=**"BACKEND"**,fallbackMethod = **"secondFallback"**)  
@GetMapping(path = **"retry"**)  
**public** ModelAndView getItems()  
{

Codeteil 8: Method mit Resilient4j Retry

##### Retries in Istio

In Istio ist es genauso einfach wie in Spring, retries für Services einzurichten. Dies geschieht über VirtualServices, welche dafür verantwortlich sind Anfragen in einem Service Mesh zu den Services zu leiten. Um retries zu benutzen werden Sidecars in den entsprechenden Pods benötigt, welche die eigentliche Wiederholung durchführen. Im Gegensatz zur Lösung in Spring werden diese Regeln auf den gesamten Service und nicht auf einzelne Endpunkte angewandt.

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

  name: myService

spec:

  hosts:

  - myService

  http:

  - route:

    - destination:

        host: otherService

        subset: v1

    retries:

      attempts: 3

      perTryTimeout: 2s

Codeteil 9: Retries durch einen VirtualService in Istio

Die Route in Codeteil 8 beschreibt ein echtes Traffic Ziel, welches im Service Mesh existiert und zu dem Anfragen gesendet werden sollen. Alles was unter retries beschrieben ist beschreibt die genauere Funktion des retry. Der Host Parameter beschreibt, auf welchen Service diese Regeln angewandt werden.

### Rate Limiting

Rate Limiting Komponenten sind Circuit Breakern ziemlich ähnlich, in der Weise, dass sie die ankommenden Anfragen limitieren. Doch anders als der Circuit Breaker wird die Auswirkung eines Rate Limiter erst ab einer bestimmten Skalierung bemerkbar und hat eine nicht so starke Auswirkung wie dieser.

"If you’ve ever worked with APIs for some huge products you know that they have rate limiting applied to almost any operation. Examples: [Facebook](https://developers.facebook.com/docs/graph-api/overview/rate-limiting), [Twitter](https://developer.twitter.com/en/docs/basics/rate-limiting), [Google Analytics](https://developers.google.com/analytics/devguides/config/mgmt/v3/limits-quotas)…” (Storozhuk 2018).

Allerdings, ganz im Gegenteil zur resultierenden Schlussfolgerung, sind die Rate Limiting Komponenten umso wichtiger für kleinere Anwendungen und es sollte vermehrt überlegt werden sie einzubauen.

Abbildung 21: Rate Limiter (Storozhuk 2018)

Rate Limiter helfen Anfragespitzen zu verhindern, was im Speziellen für kleinere Anwendungen wichtig ist. Anfragespitzen werden entweder in **Warteschlangen** abgelegt und später abgearbeitet oder einfach **abgelehnt,** (siehe Abbildung 21 die blauen Flächen werden nach hinten verschoben und später verarbeitet). Es kann nach unterschiedlichen Anfragen-Typen gefiltert werden, welche separat gehandhabt, kategorisier und unterschiedliche Limits für die entsprechenden Gruppen vorgegeben werden. Durch diese Techniken wird eine erhöhte Verfügbarkeit und Verlässlichkeit erreicht, wodurch die Anwendung darauf vorbereitet wird skalierbar zu sein. Aus diesem Grund wird Rate Limiting in vielen API Gateways oder Proxys schon direkt mitgeliefert, wie dies in Kapitel 4.1 Kommunikations-Leitung beschrieben wurde.

#### Umsetzung

##### Rate Limiting in Spring Boot2 mit Resilience4j

Um einen Rate Limiter in Spring Boot 2 mit der Resilience4j Bibliothek zu verwenden muss lediglich, wie in Codeteil 10 zu sehen ist, eine Methode oder Klasse mit der entsprechenden Annotation versehen werden. Die Annotation benötigt immer noch zusätzlich einen Namen, welcher, wie in Codeteil 11 zu sehen ist, von der application.properties Datei referenziert und eingestellt werden kann.

@RateLimiter(name=**"BACKEND"**)  
@GetMapping(path = **"resilience"**)  
**public** ModelAndView getItems()

Codeteil 10: Methode mit Rate Limiter durch Resilence4j

resilience4j.ratelimiter.instances.BACKEND.limit-for-period=4  
resilience4j.ratelimiter.instances.BACKEND.limit-refresh-period=500ms  
resilience4j.ratelimiter.instances.BACKEND.timeout-duration=2s

Codeteil 11:Einstellungen des rate Rimiter von Resilience4j

##### Rate Limiting in Istio

Rate Limiting in Istio ist ziemlich komplex und benötigt einiges an Konfiguration. Es kann zwar durchaus mehr Funktionalität erreicht werden als in Spring Boot z.B. userabhängiges Rate Limiting welches eingeloggte User nicht limitiert. Allerdings muss für das Rate Limiting zuerst die „Policy Enforcement“ für das Cluster aktiviert sein.

Die Konfiguration ist in zwei Teile aufgeteilt. Zum einen die Client Seite, welche für die Sidecars benötigt wird, zum anderen die Mixer Seite, welcher für die Policy Verwaltung und Telemetrie Sammlung verantwortlich ist. Der Mixer verteilt die entsprechenden Regeln an die Sidecars weiter, wo diese dann angewendet werden.

Auf der Client Seite benötigt es:

* QuotaSpec: Definiert wie die Quote/Rate heißt und welche Menge der Client anfragen soll.
* QuotaSpecBinding: Verknüpft den QuotaSpec mit einem oder mehreren Services.

Auf der Mixer Seite benötigt es:

* Quota instance: Definiert wie Quoten auszusehen haben.
* quota handler: ist eine Adapter, welcher entweder eine In-memory Datenbank oder eine Redis Datenbank mit entsprechendem Redis Handler verwendet. Im Handler wird zusätzlich noch das Rate Limiting für die einzelnen Endpunkte konfiguriert.
* Quota rule: Definiert wann Quoteninstanzen zum Adapter weitergeleitet werden, wodurch z.B. eine benutzerspezifische Auswahl stattfinden kann.

### Fallback

#### Was Passiert bei Fehlern?

Wenn Services ausfallen ist es die einfachste Lösung, einfach einen Fehler für den Nutzer abzusetzen. Es ist allerdings auch die wohl unschönste Antwort, welche einem Nutzer geliefert werden kann und da die Systeme in der Regel auch für Nutzer gebaut werden, sollten diese auch die höchste Priorität haben.

#### Das Fallback Muster

Das Fallback Muster ist in diesem Falle mehr eine Ergänzung zu Mustern wie Retry und Circuit Breaker, welche gewährleisten, das Services sich bei Fehlern erholen können bzw. Anfragen bei Fehlern wiederholt werden. Das Fallback Muster sorgt dafür, dass bei auftretenden Fehlern auf mögliche nützliche Informationen zurückgegriffen wird. Dabei führen sie hierbei nur wenig bis gar keine Verarbeitung durch, da das System sich bereits im Fehlerzustand befindet. Hinzu kommt, dass die entsprechenden Fallbacks nur eine geringe Wahrscheinlichkeit haben sollten selbst fehlzuschlagen.

Im Folgenden werden drei unterschiedliche Möglichkeiten vorgestellt einen Fallback einzubauen:

* Stumm Fehlschlagen: Ist die einfachste und vermutlich am weitest verbreitete Art eines Fallbacks. Hierbei wird einfach ein Nullwert zurückgegeben. Das kann dann nützlich sein wenn die angefragte Ressource optional ist und in der Antwort einfach weggelassen werden kann.
* Spezifisch/Custom Fehlschlagen: Wird in Fällen benutzt, wenn es Fallback Methoden in einer Client Bibliothek gibt. Alternativ kann versucht werden auf lokal gecachte Daten zurückgreifen, um so eine geeignete Antwort zu generieren.
* Schnelles Fehlschlagen: Wird verwendet wenn die Daten benötigt werden und es keine wirkliche alternative Möglichkeit gibt. In diesem Fall wird das wohl des Services über die der Nutzererfahrung gestellt und Erlauben dem Service sich zu erholen, sodass der Normale Verlauf wieder hergestellt werden kann.

#### Umsetzung

Die hierbei verwendete Bibliothek ist Resilient4j, die spezielle Module für das Spring Boot 2 Framework enthält, welches die Verwendung der Bibliothek deutlich vereinfacht. Eine Fallbackmethode kann hierbei mit jeglicher Robustheit Technik verknüpft werden, in diesem Beispiel ist es die Retry Technik. Die Fallbackmethode muss sich dazu in derselben Klasse befinden, wie der Aufruf (siehe Codeteil 7) und dieselbe Struktur, sowie genau einen zusätzlichen Exception Parameter besitzen (siehe Codeteil 8). Falls mehrere Fallbackmethoden angegeben wurden wird diejenige aufgerufen welche die Exception aufweist und der ausgelösten Exception am ähnlichsten ist.

@Retry(name=**"BACKEND"**,fallbackMethod = **"secondFallback"**)  
@GetMapping()  
**public** ModelAndView getItems()

Codeteil 8: Methode mit Fallback

**public** ModelAndView secondFallback(Exception e) {

cached data, alternative API, static Fallback data …

}

Codeteil 13: Aufzurufende Fallback Methode

##### Fallbacks in Istio

In Istio oder Kubernetes werden keine Fallbacks unterstützt, diese müssen vom Service selbst Implementiert werden.

### Zusammenfassung

Es wurden nun einige Probleme in verteilten System dargestellt, sowie unterschiedliche Möglichkeiten aufgezeigt, um mit diesen auftretenden Problemen zu verfahren, wodurch dargestellt wurde wie praktische Lösungen aussehen können.

Nachdem nun behandelt wurde wie Services aufrechterhalten werden können bzw. lauffähig bleiben, soll im Folgenden aufgezeigt werden wie Services überhaupt gefunden werden können, sowohl in der Theorie als auch in der Praxis und welche unterschiedlichen Möglichkeiten hierfür zur Verfügung stehen.

## Service discovery

### Einleitung

Mit der Microservice Architektur kommen anstelle von ein paar wenigen, schweren übersichtlichen und möglicherweise fest zusammenhängenden Services, sehr viele, leichte, kurzlebige und autoskalierte Services zum Einsatz (siehe Abbildung 22). Jeder dieser neuen Services soll auch die Möglichkeit besitzen untereinander kommunizieren zu können. Wo es bisher noch möglich war Services manuell zuzuweisen, Adressen festzulegen und diese manuell zu verbreiten, ist dies nun vollkommen unvorstellbar und muss automatisiert werden.

Abbildung 22: Wieso ist Service discovery notwendig (Richardson 2015)

### Was ist Service discovery

Um das beschriebene Problem zu lösen wurde die Service discovery entwickelt. Sie bietet einen oder mehrere zentrale Orte an, wo die jeweiligen gesuchten Services gefunden werden können. Bevor Services allerdings gefunden werden können, müssen sich diese zuerst anmelden. Deshalb wird diese Komponente auch Service Registrie genannt. Im selben Schritt wird in diesem Prozess auch eine Lastenverteilung durchgeführt, wodurch sie nicht mehr an anderer Stelle zusätzlich benötigt wird. Um diese Technik anzuwenden, gibt es unterschiedliche Herangehensweisen, welche nun angeschaut werden.

### Serverseitige discovery

In dieser Technik wird ein separater Load Balancer benutzt, welcher auch als Eintrittspunkt für alle Clients dient[1]. Der Load Balancer fragt bei der Service Registry nach[2] und leitet die entsprechenden Clients an die gesuchten, verfügbaren Instanzen weiter[3] (immer ein zusätzlicher Stopp).

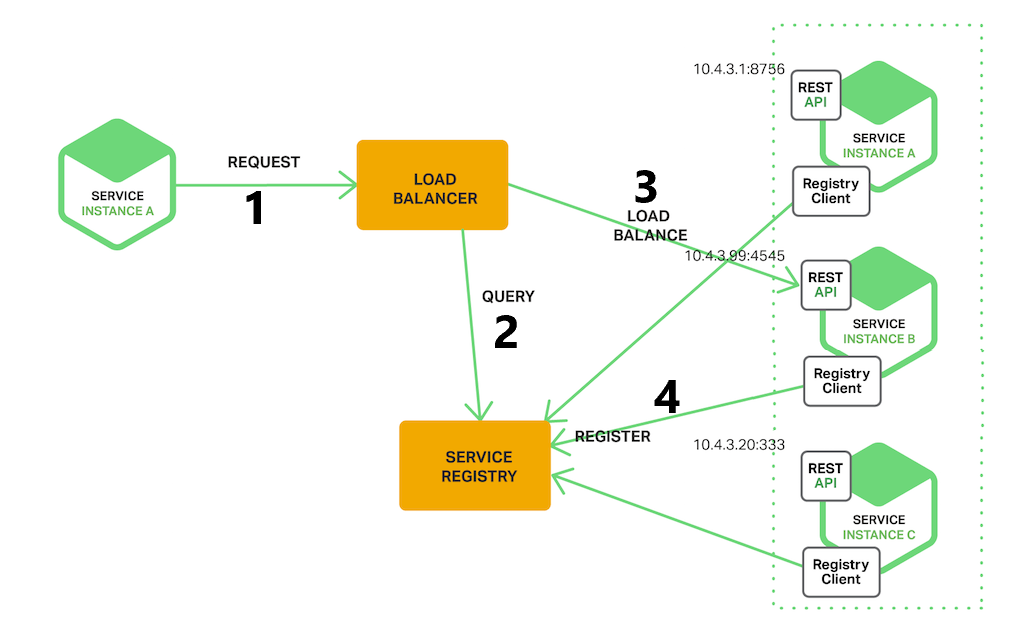


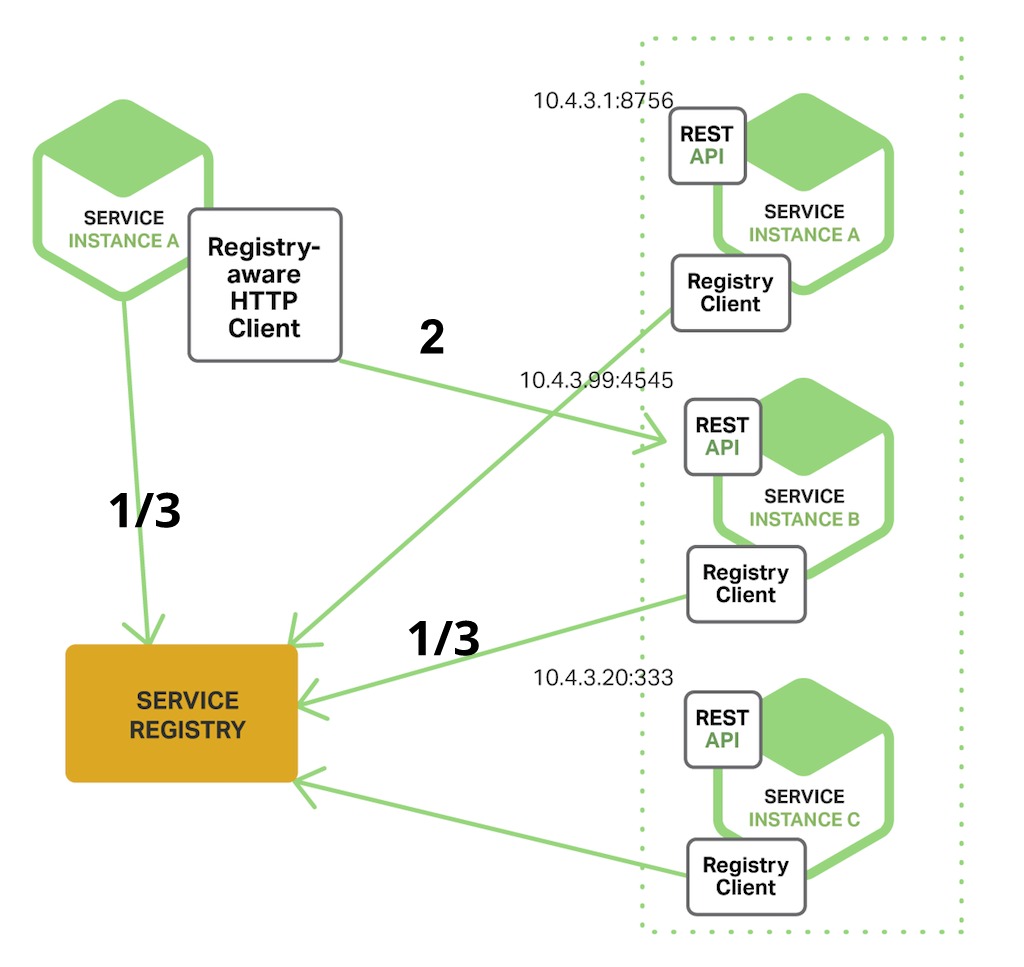
Abbildung 23: Server-Side discovery Pattern (Richardson 2015)

Auf die genauen Registrierungsverfahren [4] wird später eingegangen.

Wenn Serverseitige Discovery benutzt wird bringt dies einige Vor- und Nachteile mit sich. Durch die Abstraktion des Discovery Mechanismus ist es möglich Clients/Services unabhängig von ihrer Sprache zu benutzten und die Notwendigkeit diese zu implementieren zu entfernen. Außerdem stellen einige Bereitstellungsumgebungen (wie Kubernetes) diese bereits frei zur Verfügung. Das Problem dieser Technik ist, dass durch das Einfügen eines zusätzlichen Load Balancers, eine weitere hochverfügbare Komponente eingefügt wird.

### Clientseitige discovery

Bei der Clientseitigen Discovery Methode sind die Clients dafür verantwortlich die verfügbaren Service Instanzen zu finden und die Last über sie zu verteilen. Dafür fragen die Clients bei der Service Registry nach [1] und erhalten die verfügbaren Instanzen für die jeweilige Aufgabe. Die Clients benutzen dann einen Load Balancing Algorithmus, um einen Service auszuwählen [2].

Abbildung 24: Client-side discovery Pattern (Richardson 2015)

Auf die genauen Registrierungsverfahren wird später eingegangen [3].

Diese Methode bringt ebenso einige Vor- und Nachteile mit sich. Abgesehen von der Service Registry entstehen keine zusätzlichen, hochverfügbaren Komponenten. Die Clients können intelligente und anwendungsspezifische Load Balancing Entscheidungen durchführen. Ein auftretendes Problem betrifft die Client Registry Verknüpfung. Diese muss im Client für jede unterschiedliche Sprache bzw. jedes Framework implementiert werden.

### Selbstregistrierungsmethode

Jede Service Instanz ist in dieser Methode, selbst dafür verantwortlich sich an der Registry anzumelden (siehe eins und drei in der Abbildung 24). Außerdem sendet der Client kontinuierliche Heardbeats/Nachrichten, um nicht abgemeldet zu werden. Die Nachteile gleichen denen der Clientseitigen Methode. Der Registrierungscode muss für jede Sprache oder jedes Framework erneut geschrieben werden wenn diese nicht schon vorhanden sind.

### Drittparteiregistrierungsmethode

Bei dieser Methode sind die Clients/Services nicht selbst verantwortlich sich an- oder abzumelden. Stattdessen übernimmt diese Aufgabe eine Drittkomponente namens Service Registrator. Der Registrator überwacht Änderungen der Services entweder durch Heardbeats oder durch Überwachung der Events des Services (Abbildung 23: Server-Side Discovery Pattern (Richardson 2015)[4]). Wenn der Registrator neue Services erkennt, Registriert er diese an der Registry und meldet diese wieder ab, sollten diese nicht länger verfügbar sein.

Der große Vorteil eine Drittkomponente zu verwenden, besteht darin, dass die Services nicht mehr eng mit der Registry verbunden sind. Es ist nicht notwendig für die selbst Registrierung eine Implementierung für jeden unterschiedlichen Service zu schreiben. Der Nachteil der dabei entsteht ist, dass eine weitere hochverfügbare Komponente hinzukommt welche gemanagt werden muss.

### Umsetzung

#### Beispiel für Clientseitige Discovery mit Selbstregistrierungsmethode

Anhand von Eureka, welche von Netzflix gebaut wurde, wird als Nächstes eine Clientseitige Discovery mit Selbstregistrierung aufgezeigt, wobei sowohl Server als auch Client vorgestellt wird. Das zu Grunde liegende System ist Java mit dem Spring-Boot Framework für welches Eureka speziell gebaut wurde. Wie bei allen anderen Bibliotheken wird auch für Eureka eine Abhängigkeit über z.B. Maven benötigt (Maven managet unter anderem die Abhängigkeiten in einem Java Projekt).

Der Discovery Server:

Der Discovery Server ist, wie zu sehen ist, dank Eureka und Spring Boot äußerst schlank und übernimmt fast alle Arbeit. Alleine die Notation @EnableEurekaServer wird benötigt um den Server zu starten. Jegliche Einstellungen für den laufenden Server werden über die application.properties Datei eingestellt.

@EnableEurekaServer  
@SpringBootApplication  
**public class** DiscoveryServerApplication {

Codeteil 10: Service discovery Server

Der discovery Client:

Der Service discovery Client benötigt hierbei schon etwas mehr Code von der Entwickler Seite (siehe Codeteil 10). Zum Anfang wird, wie beim Server auch, die @EnableDiscoveryClient Notation benötigt. Als nächstes können einige Variablen gesehen werden, welche aufzeigen wie die bisherige hartgecoded Verbindung aussah. Durch die @Autowirded markierte DiscoveryClient Variable wird diese automatisch initialisiert und kann im Anschluss verwendet werden. Die folgende Methode übernimmt den Namen einer Applikation, welche gesucht wird und fragt anschließend über den Initialisierten DiscoveryClient bei der Registry mit dem entsprechenden Service Namen nach. Im Anschluss würde nun das clientseitige Loadbalancing stattfinden, wobei in diesem Falle eine beliebige Instanz verwendet wird, um im Anschluss dessen Host aufzulösen und zurückzugeben. Weitere spezifische Einstellungen für den Client, werden wie auch beim Server über die application.properties Festgelegt.

Die Registrierung am Server geschieht dabei vollkommen automatisch im Hintergrund.

@SpringBootApplication  
@EnableDiscoveryClient  
**public class** TodoUiApplication {  
 String **backendHost**=**"localhost"**;  
 String **backendPort**=**"8080"**;

String **host**= **""**;  
 @Autowired  
 **private** DiscoveryClient **discoveryClient**;  
 @RequestMapping(**"/service-instances/{applicationName}"**)  
 **public** String serviceInstancesByApplicationName(@PathVariable String applica tionName){  
 List<ServiceInstance> nameSpecificInstances = **this**.**discoveryClient**.getInstances(applicationName);  
 Optional<ServiceInstance> optionalServiceInstance = nameSpecificInstances.stream().findAny();  
 **if**(optionalServiceInstance.isPresent())  
 {  
 **host** = instances.stream().findAny().get().getUri().toString();  
 }  
 System.***out***.println(**host**);  
 **return host**;  
 }

Codeteil 11: Service discovery Client

#### Drittparteiregistrierungsmethode

Beispiel anhand von Istio(Kubernetes):

Istio selbst besitzt keinen eigenen Service Discovery Mechanismus, dafür allerdings eine Service Registry. Für den Discovery Mechanismus greift Istio auf das unterliegende Orchestrierungssystem zurück, wobei hier sehr oft Kubernetes zur Anwendung kommt und aus diesem Grund auch hier verwendet wird.

Nun ein einfaches Beispiel eines Services in einem Namensraum in Kubernetes welches mit dem DNS Add-on ausgestattet ist und seit Version 1.11 standardgemäß mit kubeadm mitgeliefert wird.

Zuerst wird der Namensraum benötigt:

apiVersion: v1

kind: Namespace

metadata:

  name: namespacename

Codeteil 16: Kubernetes Namensraum

Als nächstes wird ein Deployment verfasst, welches eine App erstellt, dieser einen internen Port und einen Namen zuweist, sowie mehrere Kopien der App erstellt.

apiVersion: v1

kind: Deployment

metadata:

  name: service-deployment

  namespace: namespacename

spec:

  replicas: 2

  selector:

    matchLabels:

      app: myappname

  template:

    metadata:

      name: somename

      labels:

        app: myappname

    spec:

      containers:

      - name: myappname

        image: gerry1313/myservice:0.5.0

        ports:

        - containerPort: 8080

Codeteil 17: Kubernetes Deployment

Zuletzt wird ein Service benötigt, welcher die nach außen offenen Ports für die App festlegt.

apiVersion: v1

kind: Service

metadata:

  name: myservicename

  namespace: namespacename

spec:

  ports:

    - port: 80

      targetPort: 8080

  selector:

    app: myappname

Codeteil 18: Kubernetes Service welcher die App verfügbar macht

Das DNS add-on stellt nun sicher, dass der Service myservicename verfügbar ist und über folgenden Namen erreicht werden kann:

myservicename.namespacename.svc.cluster.local

<service>.<ns>.svc.<zone>. <ttl> IN A <cluster-ip>

Codeteil 19: Kubernetes DNS Name

Die zweite Zeile im Codeteil 19, beschreibt die Semantik wie ein solcher DNS Name aufgebaut wird. Hierbei steht <zone> für die Cluster domain, <ns> für den Namensraum, <ttl> für die time to live und <cluster-ip> die IP unter welcher das Cluster zu erreichen ist.

# Zusammenfassung

Diese Arbeit hat sich besonders mit der Kommunikation in einem verteilten System von Microservices beschäftigt und für diesen Zweck unterschiedliche Techniken und Möglichkeiten untersucht. Es galt hierbei sich verschiedener Probleme anzunehmen, welche in diesem Umfeld auftreten und unterschiedliche Lösungsmöglichkeiten zur Verfügung zu stellen. Außerdem war es für das Verständnis wichtig zu verstehen, wo diese Probleme entstehen und dass diese nicht einfach vermieden werden können.

Zur Einleitung in die Thematik wurden zuerst Microservices erläutert und ein Vergleich zwischen einer Monolithischen und Microservice Architektur aufgezeigt, welcher dem Leser dabei hilft, eine Entscheidung für eigene Projekte und Anwendungen zu treffen. Hierbei konnten sowohl die Vor- als auch die Nachteile beider Ansätze erkannt und berücksichtigt werden. Als Nächstes wurde der betreffende Themenbereich auf verteilte Systeme erweitert, um sich in einem reelleren Umfeld zu bewegen. Infolgedessen wurden weitere Probleme erläutert, welche durch diesen neuen Themenbereich hinzukamen und Systeme vorgestellt, welche dabei helfen diese Probleme zu bewältigen und später weiterverwendet werden sollten.

Im Hauptteil dieser Arbeit, wurde zuerst das Problem der Kommunikationsleitung angegangen und zwei primäre Herangehensweisen untersucht, welche zum einen durch einen Serverseitigen API Gateway widergespiegelt wird und zum anderen durch einen Clientseitigen API Microgateway. Hierbei hat sich das Microgateway, welches auch als Proxy oder Sidecar bezeichnet wird, im Bereich der Service Meshes zum Standard etabliert und kann als Weiterentwicklung angesehen werden. Das API Gateway wiederum ist ein Hauptbestandteil eines API Managements und ist ein zentraler Knotenpunkt in diesem System.

Management von APIs kann zu einem immer größeren Problem werden, welches durch ein API Management System angegangen werden kann. Um die Stärken und Unterschiede zu einem Service Mesh zu verstehen wurde hierfür noch ein Vergleich dargestellt, welcher dabei hilft ein API Management einzuordnen. Zur Übersicht und damit eine Möglichkeit der Überwachung von Services zu erhalten, wurde ein kleiner Einblick in Distributed Tracing gegeben, um auch hierfür ein Werkzeug zur Hand zu haben.

Microservices in verteilten Systemen, besitzen ein klares, erhöhtes Fehlerpotential. Es wurde aufgezeigt, wie diese unterschiedlichen Fehler aussehen und mögliche Techniken vorgestellt um diese zu meistern. Für jede Technik, wurde hierbei möglichst eine Lösung im Bereich der Anwendungsprogrammierung und der Infrastruktur gesucht, um alternative Herangehensweisen zu ermöglichen. Die Techniken erstrecken sich dabei über Bulkheads, Circuit Breaker, Retries, Rate Limits und Fallbacks, welche je eigene Probleme behandeln. Für Anwendungen wurde dabei Resilient4j in Verknüpfung mit Spring Boot 2 verwendet und für die Infrastruktur das Service Mesh Istio bzw. die unterliegende Orchestrierungsplattform Kubernetes. Es ist hierdurch möglich zu verstehen wie mit entsprechenden Problemen umgegangen werden kann und wo am besten eine entsprechende Lösung eingebaut werden sollte.

Um in einem dynamischen, sich verändernden verteilten System Kenntnis über den Aufenthaltsort, der vorhandenen Services zu erhalten, benötigt es einer Lösung. Hierfür wurde sich mit der Service Discovery beschäftigt und unterschiedliche Herangehensweisen aufgezeigt, sowie mögliche Umsetzungen für aktuelle Systeme wie Spring Boot oder Istio bzw. Kubernetes dargestellt. Durch eine immer verbreiterte Verwendung von Kubernetes, ist diese Lösung im Bereich der verteilten Systeme immer relevanter, eine Alternative für diesen oder kleinere Bereiche ist allerdings ebenso notwendig.

Diese Arbeit bietet einen guten Überblick über aktuelle Themenbereiche im Feld der verteilten Systeme von Microservices im Zusammenhang mit Vergleichen zu bisher dagewesenen Technologien. Sie hilft dabei Entscheidungen zwischen unterschiedlichen Herangehensweisen zu Treffen und schafft ein Bewusstsein für mögliche Stolpersteine bzw. Fehlerpotentiale. Durch unterschiedliche Möglichkeiten im Bereich der Kommunikationssteuerung und -aufbau und der Kontrolle bietet sie Einblicke in diese Bereiche und führt den Leser an diese Themen heran.

# Ausblick

Die Welt, soweit man das momentan sagen kann, wird auch in Zukunft immer globaler. Verteilte Systeme und Microservices werden immer mehr Verwendung finden, um global verteilte Probleme zu lösen. Die notwendigen Werkzeuge und Verständnisse der entsprechenden Techniken werden dadurch immer wichtiger. Ein Überblick über die unterschiedlichen Möglichkeiten bei einer Vielzahl an Anbietern wird dabei immer schwieriger.

Einige weitere Möglichkeiten, mit denen sich in diesem Umfeld beschäftigt werden könnte und nicht im Umfang dieser Arbeit Platz fanden, werden im Folgenden hervorgehoben.

Der Aspekt der Sicherheit, speziell im Umfeld von Service Meshes, sollte in einer eigenständigen Arbeit erörtert werden und könnte Lesern eine bessere Einschätzung dieser Technologie verschaffen.

Die Datenbeständigkeit, welche in verteilten Systemen fast immer zu Schwierigkeiten führt, wäre im Bereich der Service Meshes und den damit verbundenen Möglichkeiten, ebenfalls einen Blick wert.

Zuletzt wäre eine praktische Umsetzung eines API Management Systems im Zusammenspiel mit einem Service Mesh interessant, da auch auf diese aus zeitlichen Gründen nicht tiefer eingegangen werden konnte, aber dennoch von großer Bedeutsamkeit wäre. Hierbei könnten die in dieser Arbeit erwähnten Systeme verwendet werden, welche zum einen das Service Mesh Istio und zum anderen das API Management Gateway wso2 wäre.

# Ehrenwörtliche Erklärung

|  |  |  |  |
| --- | --- | --- | --- |
| Name: | Wildermuth | Vorname: | Gerrit |
| Matrikel-Nr.: | 749442 | Studiengang: | SWB |

Hiermit versichere ich, Gerrit, Wildermuth, dass ich die vorliegende Bachelorarbeit mit dem Titel Möglichkeiten zur Kontrolle und Steuerung der Kommunikation von Microservices selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebene Literatur und Hilfsmittel verwendet habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken ent­nommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Fellbach, 27.2.2020

Ort, Datum Unterschrift

Literaturverzeichnis

Dashora, Saurabh: *The Rise of Spring Framework | PROGRESSIVE CODER.* URL http://progressivecoder.com/the-rise-of-spring-framework/. – Aktualisierungsdatum: 16.01.2019 – Überprüfungsdatum 13.12.2019

Dimensional Research: *global-microservices-trends-2018.* A SURVEY OF DEVELOPMENT PROFESSIONALS (2018). URL https://go.lightstep.com/rs/260-KGM-472/images/global-microservices-trends-2018.pdf – Überprüfungsdatum 27.01.2020

Eigenkreation

Fowler, Martin: *CircuitBreaker.* URL https://martinfowler.com/bliki/CircuitBreaker.html. – Aktualisierungsdatum: 10.04.2019 – Überprüfungsdatum 14.10.2019

Golden, Bernard CEO Navica: *3 reasons you should always run microservices apps in containers | Learn.* URL https://learn.techbeacon.com/units/3-reasons-you-should-always-run-microservices-apps-containers. – Aktualisierungsdatum: 05.02.2017 – Überprüfungsdatum 22.11.2019

IBM: *Limiting the number of concurrent requests to microservices.* URL https://openliberty.io/guides/bulkhead.html#background-concepts. – Aktualisierungsdatum: 28.11.2019 – Überprüfungsdatum 12.02.2019

Kubernetes: *Kubernetes Deployments.* URL https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment – Überprüfungsdatum 25.11.2019

Kyma: *Tracing - Docs | Kyma - An easy way to extend enterprise applications on Kubernetes.* URL https://kyma-project.io/docs/components/tracing#details-benefits-of-distributed-tracing – Überprüfungsdatum 11.07.2019

Layer5: *Service Mesh Landscape.* URL https://layer5.io/landscape/. – Aktualisierungsdatum: 28.10.2019 – Überprüfungsdatum 30.10.2019

McAllister, Dave: *GOTO 2019 • Observability, Distributed Tracing & the Complex World • Dave McAllister - YouTube.* URL https://www.youtube.com/watch?v=2nTJSsBngao – Überprüfungsdatum 11.08.2019

Microsoft, dragon119: *Wiederholungsmuster - Cloud Design Patterns.* URL https://docs.microsoft.com/de-de/azure/architecture/patterns/retry – Überprüfungsdatum 12.03.2019

Paraschiv, Eugen: *The State of Java in 2018.* URL https://www.baeldung.com/java-in-2018#spring-adoption. – Aktualisierungsdatum: 11.06.2018 – Überprüfungsdatum 13.12.2019

Posta, Christian: *API Gateways Are Going Through an Identity Crisis.* URL https://blog.christianposta.com/microservices/api-gateways-are-going-through-an-identity-crisis/. – Aktualisierungsdatum: 116/0/2019 – Überprüfungsdatum 12.03.2019

Prinz, Hanna: *Service Mesh – für Microservices unverzichtbar?* URL https://www.informatik-aktuell.de/entwicklung/methoden/service-mesh-fuer-microservices-unverzichtbar.html. – Aktualisierungsdatum: 15.10.2019 – Überprüfungsdatum 12.05.2019

Richardson, Chris: *Microservices Pattern: API gateway pattern.* URL https://microservices.io/patterns/apigateway.html. – Aktualisierungsdatum: 20.11.2019 – Überprüfungsdatum 12.02.2019

Richardson, Chris: *Service Discovery in a Microservices Architecture - NGINX.* URL https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ – Überprüfungsdatum 31.10.2019

Rotem-Gal-Oz, Arnon: *Fallacies of distributed computing Explained.* URL http://www.rgoarchitects.com/Files/fallacies.pdf – Überprüfungsdatum 10.09.2019

Sebastian Eschweiler: *Docker – Beginner’s Guide – Part 1: Images & Containers.* URL https://codingthesmartway.com/docker-beginners-guide-part-1-images-containers/. – Aktualisierungsdatum: 24.02.2019 – Überprüfungsdatum 26.01.2020

StackRox: *The State of Container and Kubernetes Security Spring 2019 : With observations and analysis from AimPoint Group.* 2019

Storozhuk, Bogdan: *Rate Limiter Internals in Resilience4j.* URL https://medium.com/@storozhuk.b.m/rate-limiter-internals-in-resilience4j-48776e433b90#7585 – Überprüfungsdatum 21.10.2019

Tanenbaum, Andrew S.; Maarten Van Steen (Mitarb.): *Distributed Systems Principles and Paradigms.* Zweite Auflage : Pearson Education, 2006

Ushio, Tsuyoshi: *Kubernetes in three diagrams.* URL https://medium.com/@tsuyoshiushio/kubernetes-in-three-diagrams-6aba8432541c. – Aktualisierungsdatum: 05.02.2018 – Überprüfungsdatum 22.11.2019

Vogels, Werner: *A Conversation with Werner Vogels - ACM Queue.* URL https://dl.acm.org/doi/pdf/10.1145/1142055.1142065. – Aktualisierungsdatum: 30.06.2006 – Überprüfungsdatum 21.11.2019