Möglichkeiten zur Kontrolle und Steuerung der Kommunikation von Microservices

**Bachelorarbeit**

im Studiengang  
Softwaretechnik und Medieninformatik

vorgelegt von

**Gerrit Wildermuth**Matr.-Nr.: 74734

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. Bei diesem wird sich im Bereich der Steuerung genauer mit der Thematik der Leitung und Management der Kommunikation beschäftigt als 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. Bevor es allerdings in den Hauptteil geht, wird sich zuerst mit der Thematik von Microservices im Allgemeinen auseinandergesetzt, sowie eine Einführung in weitere Themenbereiche der verteilten Systeme, welche für ein späteres Verständnis benötigt werden.

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. Hierbei kamen allerdings auch einige Probleme mit sich, welche sich unter anderem in einer erhöhten Kommunikation auswirkte. Um schneller auf Lastenunterschiede reagieren zu können und weniger Ressourcen zu verbrauchen wird hierbei vermehrt auf Microservices gesetzt. Um hierbei einen Überblick zu bekommen welche Techniken und Bausteine hierbei benötigt werden oder Hilfreich sind um Systeme zu gestalten, welche mit dieser Problematik umgehen, wurde sich mit dieser Thematik befasst 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 taken 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

Abkürzungsverzeichnis 7

1 Überblick 8

2 Microservices 10

2.1 Wieso Microservices 10

2.2 Monolith 10

2.2.1 Einführung 10

2.2.2 Vorteile der Monolithen 11

2.2.3 Nachteile der Monolithen 11

2.3 Microservices 12

2.3.1 Einführung 12

2.3.2 Vorteile der Microservices 12

2.3.3 Nachteile der Microservices 13

3 Verteilte Systeme 15

3.1 Definition 15

3.2 Orchestrierung 15

3.3 Netzwerk/Kommunikation in einem Verteilten System 15

3.4 Von Fehler Potential zu Robustheit 16

3.5 Daten Beständigkeit 16

3.6 Einführung in Kubernetes 16

3.6.1 Kubernetes Komponenten 17

3.6.2 Kubernetes Architektur 20

4 Möglichkeiten zum Management und Gewährleisten eines verteilten Systems 22

4.1 Kommunikations-Leitung 22

4.1.1 API Gateway 22

4.1.2 API Microgateway 24

4.2 Gewährleistung von Services 24

4.2.1 Einleitung 24

4.2.2 Bulkhead 25

4.2.3 Circuit breaker 27

4.2.4 Retry 30

4.2.5 Timeout? 31

4.2.6 Rate Limiting 31

4.2.7 Fallback 32

4.3 Kommunikations- Überwachung und Verfolgung 32

4.3.1 API Management 32

4.3.2 Distributed tracing 34

4.4 Service discovery 35

4.4.1 Wo sind alle Services? 35

4.4.2 Was ist Service discovery 35

4.4.3 Serverseitige discovery 36

4.4.4 Clientseitige discovery 37

4.4.5 Selbstregistrierungs Methode 38

4.4.6 Drittparteiregistrierungs Methode 39

4.5 Vereinheitlichte Systeme 39

4.5.1 Service-Mesh 39

4.5.2 Spring Cloud 41

4.6 Status und Daten Verbreitung in verteilten Systemen 43

4.6.1 Daten Konsistenz 43

4.6.2 Konfiguration Server 43

5 Umsetzung eines robusten verteilten Systems mit Hilfe eines Service-Meshes 44

5.1 Das Service-Mesh Istio 44

5.1.1 Istio Architecture 44

5.1.2 Istio Notizen 44

6 Notizen 45

7 Mögliche Quellen 47

Zusammenfassung und Ausblick 48

Glossar 49

Ehrenwörtliche Erklärung 50

Stichwortverzeichnis 51

8 Literaturverzeichnis 52

# 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: Container deployment in Kubernetes (Ushio 2018) 17

Abbildung 4: Kubernetes Komponenten und deren Interaktion (Ushio 2018) 18

Abbildung 5: Deplyoment Yaml (Kubernetes) 19

Abbildung 6: Kubernetes Architektur Übersicht (Ushio 2018) 20

Abbildung 21: Nutzung von Bibliotheken VS Auslagerung in Sidecars (Prinz 2019) 22

Abbildung 22: Service Mesh Architektur (Prinz 2019) 23

Abbildung 23: Spring Annahme (Paraschiv 2018) 24

Abbildung 7: API Gateway Übersicht (Richardson 11/20/2019) 27

Abbildung 8: API Gateways Backends for frontends (Richardson 11/20/2019) 28

Abbildung 9: Microservices ohne den Bulkhead (Eigenkreation) 29

Abbildung 10: Schiff Links Ohne Bulkhead (IBM 11/28/2019) 30

Abbildung 11: Schiff Rechts mit Bulkhead (IBM 11/28/2019) 30

Abbildung 12: Microservices mit dem Bulkhead (Eigenkreation) 31

Abbildung 13: Circuit Breaker Zustände (Fowler 2014) 34

Abbildung 14: Retry Pattern (Microsoft) 36

Abbildung 14: Rate Limiter (Storozhuk 2018) 38

Abbildung 15: API Managment Gateway Overview (Posta 2019) 40

Abbildung 16: Distributed Tracing (Kyma) 41

Abbildung 17: Wiso ist Service discovery notwendig (Richardson 2015) 42

Abbildung 18: Server-Side discovery Pattern (Richardson 2015) 43

Abbildung 19: Client-side discovery Pattern (Richardson 2015) 44

# Tabellenverzeichnis

Tabelle 1: Unterschiede zwischen Abbildungen und Tabellen 27

Tabelle 2: Beispiele für Überschriftebenen 36

Tabelle 3: Aufstellung der wichtigsten Formatvorlagen der Dokumentvorlage 48

# Codeverzeichnis

[Codeteil 1: Bulkhead Serverseitige Implementierung in Java mit resilience4j in Plain Java 32](#_Toc30959309)

[Codeteil 2: Bulkhead Fehlermeldung wenn der Threadpool vollgelaufen ist. 32](#_Toc30959310)

[Codeteil 3: Resilient4j mit Spring Boot 2. 32](#_Toc30959311)

[Codeteil 4: Methode welche mit Resilient Annotationen des Circuit breakers versehen ist. 35](#_Toc30959312)

[Codeteil 5: Circuit breaker Einstellungen in der application.properties 35](#_Toc30959313)

[Codeteil 9: Method mit Resilient4j Retry 37](#_Toc30959314)

# Abkürzungsverzeichnis

ALP Arbeits-, Lern- und Präsentationstechniken

HBI Hochschule für Bibliotheks- und Informationswesen

HdM Hochschule der Medien

HSE Hochschule Esslingen

# 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 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 wie dies bei bisherigen Systemen der Fall war. Dieser Kommunikationsanstieg kann dazu führen das deutlich mehr Fehler auftreten, sich deutlich mehr um den Kommunikationsfluss gekümmert werden muss, es nicht mehr möglich ist manuell einzelne Serviceadressen weiterzuverteilen 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 werden hierbei unterschiedliche Möglichkeiten geboten, 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 die Kommunikation. Um die Kommunikation zu ermöglichen, benötigen wir Kenntnis über unsere Services im Speziellen über ihren Aufenthaltsort. Damit wir diesen in einem sich möglicherweise schnell verändernden System noch finden können benötigen wir eine dynamische Lösung zum Lösen dieses dynamischen Problems.

Nachdem wir Kenntnis über den Aufenthaltsort unserer Services haben muss der Kommunikationsfluss in unserem verteilten System wieder ermöglicht bzw. hergestellt werden. Hierfür wird sich mit Bausteinen beschäftigt, welche helfen sich mit diesem Problem zu befassen und es möglich machen unsere Anfragen durch unser System durchzuleiten, um sie genau dort hinzuführen, wo sie hingehören.

Nachdem der Kommunikationsfluss wiederhergestellt ist und die APIs wieder zugänglich sind, wird es daraufhin wichtig zu kontrollieren, wer Zugriff auf unser System hat. Sowohl Nutzer als auch Entwicklern benötigen eigene Möglichkeiten sich mit unserem 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, wobei wir uns hierbei auf die Programmiersprache Java beschränken um den Rahmen nicht zu sprängen. Die hierbei Verwendung findenden Umgebungen sind zum einen Spring Boot/Cloud, als auch das Service Mesh Istio welche beide im Verlauf der Arbeit noch erklärt werden.

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 Richtig für das entsprechende Problem ist, sowie einige weitere Probleme aufzeigt welche hiermit einherkommen können.

# Microservices

Im folgendem wird sich mit Microservices befasst und Erklärt woher sie kommen worauf sie Aufgebaut sind und eine Einführung in die Thematik dieser gegeben. Des Weiteren wird eine Gegenüberstellung mit einem Monolithischen System geben wodurch einige weitere vor und Nachteile dieser beiden Systeme veranschaulicht werden um dem Leser die Möglichkeit zu geben selbst eine Entscheidung zu treffen welches eher das richtige System für seinen Zweck 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. “The many things that you would like to see happening in a good software environment couldn’t be done anymore; (Vogels 6/30/2006). Diese Systeme konnten ab einem Zeitpunkt/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 6/30/2006). 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. Die Microservice Architektur ist hierbei eine solcher Architekturen welche es ermöglicht diese Punkte umzusetzen und ist ebenso eine sehr stark wachsende Architektur „Microservices have become mainstream in the enterprise“ (Dimensional Research 2018).

## Monolith

### Einführung

Ein Monolith ist im Deutschen ein einzelner Stein. In unserem 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 aber das System wird 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 andern 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 und somit kann auf weitläufige Kommunikation verzichtet werden und einiges an Fehlern und Arbeitsaufwand vermieden werden. Es ist dadurch 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 ist an einem Punkt wodurch Daten Konsistenz deutlich einfach bzw. überhaupt erreicht werden kann.

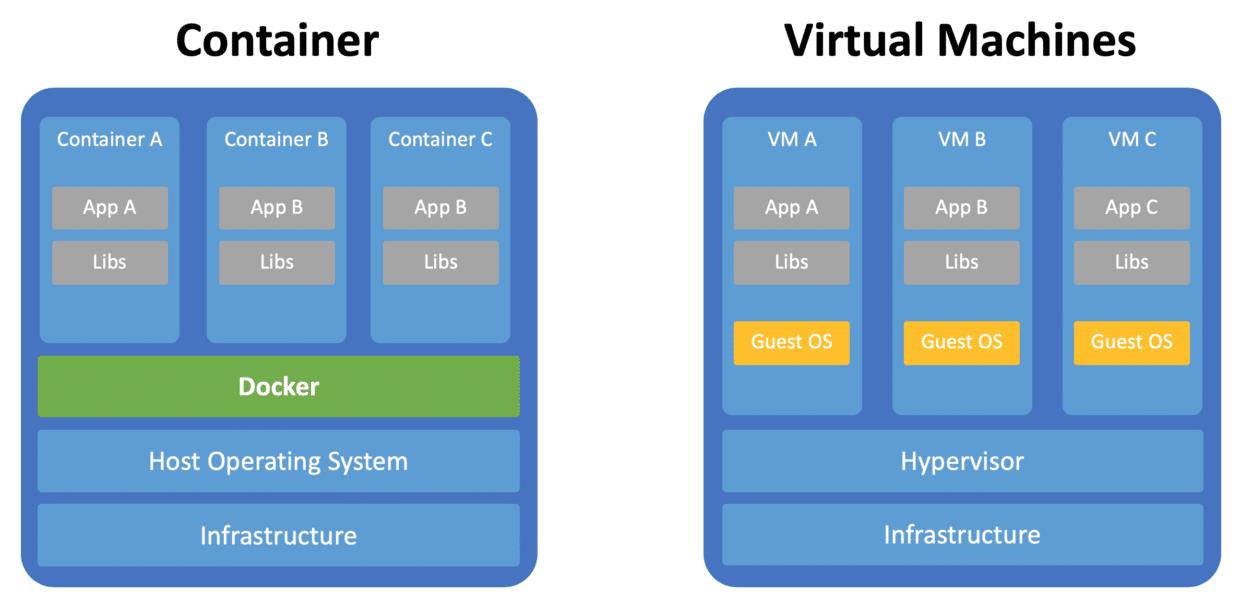
### Nachteile der Monolithen

* 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 gleichzeitig an vielen Orten 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 ausgelagert in eigene Services. 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 Monolith VS Microservices in Abbildung 1 in viele einzelne Datenbanken aufgeteilt. Es hat sich hierbei etabliert, dass Microservices für gewöhnlich in Containern auftreten. „The best choice for running a microservices application architecture is application containers.” (Golden 5/2/2017) Container bilden hierbei ein Gehäuse für eine entsprechende Anwendung mit all ihren Abhängigkeiten, Laufzeitumgebungen, System Werkzeugen, System Bibliotheken und Einstellungen (Abbildung 2). Die Container bilden isolierte Bereiche dar mit ihren 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 5/2/2017)

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 Entwicklern 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.
* 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 sind und dadurch über das Netzwerk getrennt wird es bei größeren Systemen deutlich schwieriger Fehler zu finden. Dies wird dadurch noch verstärkt, wenn die Services von unterschiedlichen Teams entwickelt werden.
* Wenn man Services baut kann die Unterteilung der logischen Prozesse zu weit gehen, wodurch diese zu sehr auseinander gezerrt werden was die Nützlichkeit der Verteilung überschreiten 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 z.B. sie nicht noch weitere Technologien benutzen möchten oder das Know-how fehlt.
* Mit vielen Abhängigkeiten zwischen den Services, kann es herausfordernd werden, ein 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.
* Datenbank Anfragen, 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 ist).

# Verteilte Systeme

## Definition

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

## Orchestrierung

Es wird besonders wichtig, wenn man ein verteiltes System mit einer Microservice Architektur verwirklicht, dass durch die deutlich größere Anzahl an Services und die dadurch entstehenden Schnittstellen diese gut gemanagt werden können. Durch Containerisierung und Autoskalierung können die Services sich deutlich schneller vermehren was dazu führt, das es schnell unübersichtlich wird. Für diese deutlich größere Anzahl an Services benötigen wir neue Werkzeuge, um diesem Wachstum zu beherrschen und möglichst zu automatisieren. Diese Werkzeuge nennt man in der Fachsprache Orchestratoren und werden dafür benutzt diese Container automatisiert zu konfigurieren, koordinieren und zu managen. Sie unterscheiden sich hierbei natürlich in Funktionalität und Handhabung. In unseren Untersuchungen, werden wir zu der Orchestrierungs Plattform Kubernetes zurückgreifen, da sie zum Zeitpunkt der Thesis die am weitesten verbreitete Plattform ist und diese wohl auch noch eine Weile sein wird.

## Netzwerk/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 hundert Prozentige Sicherheit. Durch diese Limitationen werden unterschiedliche Technologien benötigt und angewendet. Es können z. B. verzögerte Wiederholungen ausgeführt werden, Teilinformationen abgerufen oder gar noch gecachte Informationen weiter verwendet werden.

## Von Fehler Potential zu Robustheit

Zu aller Erst geht es nicht darum falls oder ob etwas Fehlschlägt, sondern wenn etwas Fehlschlägt. Das heißt, wir rechnen fest damit, dass Fehler auftreten werden und Planen für diese im Voraus. In verteilten Systemen trifft dies besonders stark zu, da die erhöhte Anzahl und Vielfalt an Kommunikation, über das Netzwerk, geradezu dazu einlädt. Doch, da wir uns dessen Bewusst sind, können wir entsprechende Komponenten und Sicherheitsmechanismen in unterschiedlichen Schichten einbauen, um diese Systeme Robuster zu machen.

Einige Fehlerquellen:

1. Es können Fehler im Netzwerk auftreten.
2. Die Netzwerkbandbreite kann überschritten werden.
3. Anfrage Wiederholungen können Lasten noch vergrößern.
4. Nachrichten Warteschlangen 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)

Einigen dieser Fehlerquellen werden später noch angeschaut und behandelt. Hierbei werden unterschiedliche Lösungsansätze geprüft und eingeordnet.

## Daten Beständigkeit

Bei einem Monolithischen System ist es üblich, dass es eine zentrale Datenbank gibt, wo jeglicher Datenverkehr gebündelt wird. In einem verteilten System wird ein anderer Ansatz verwendet und jedes System verwaltet eigene Datenbanken. 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 verschieden Datenbank Systeme, in einem einzigen System zu verwenden.

## Einführung in Kubernetes

Wie bereits zuvor im Thema 3.2 Orchestrierung erwähnt, wird zur Durchführung dieser Thesis die Orchestrierungsplattform Kubernetes verwendet. Im Folgenden werden die einzelnen Komponenten von Kubernetes erklärt um ein grobes Verständnis darüber zu erlangen wie sie Funktionieren und miteinander Interagieren.

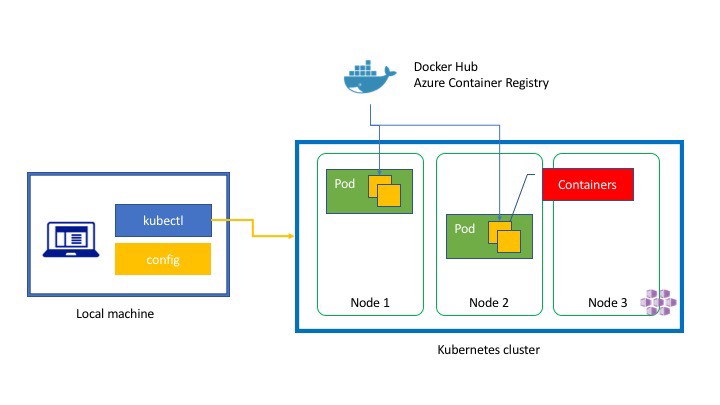


Abbildung 3: Container deployment in Kubernetes (Ushio 2018)

Um Container in Kubernetes zu laden sodass diese dort Verwendet werden, wird eine Yaml Datei benötigt welche diese spezifiziert und Unterschiedliche Parameter für diese Festlegt wie z.B. den Namen, den Port, das zu verwendende Image …Durch die Yaml Datei weiß Kubernetes um welche Docker/Azure Images es sich handelt und bezieht bzw. Konfiguriert diese Automatisch. Für diesen Prozess wird die Kubectl verwendet welche lokal Eingerichtet ist und in einer Konfigurationsdatei die nötigen Informationen für das momentane Cluster gespeichert hat. Die Kubectl wird für die Interaktion mit dem Cluster verwendet und ist die Schnittstelle für dieses.

### Kubernetes Komponenten

#### Pods

Ein Pod ist immer Bestandteil eines Nodes und werden Automatisch über diese Verteilt. Pods können einen oder mehrere Container beinhalten. (Abbildung 3: Container deployment in Kubernetes (Ushio 2018)) Jeder Container kommt mit einer Spezifikation welche für die Repräsentation und Einstellungen im Cluster beinhaltet. Falls Pods mehrere Container beinhalten, sind diese relative eng miteinander verbunden und befinden sich im selben Netzwerk so wie auch einen möglichen Zugriff auf den gleichen Speicher.

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

#### Replica Set

Das Replica Set stellt sicher das immer eine festgelegte Anzahl an Pods, eines Typs vorhanden und Verfügbar ist (Orangene Linien in Abbildung 4: Kubernetes Komponenten und deren Interaktion (Ushio 2018)). Jeder Pod wird außerdem mit einer owner Referencer ausgestattet wodurch diese Konkreten Replica Sets zugewiesen werden. Da die Replica Sets für gewöhnlich von den Deployments verwaltet und benutzt werden, kann es sein das man diese niemals direkt Manipulieren muss, sondern dies durch die Spezifikation der Deployments schon getan hat.

#### Deployment

Abbildung 4: Kubernetes Komponenten und deren Interaktion (Ushio 2018)

Ein Deployment bestimmt welche Pods und Replica Sets Erstellt werden und beschreibt welchen erwünschten Status diese haben. Der Deployment Controller führt hierbei die eigentliche Arbeit aus. Der Controller verändert das Deployment auf eine Kontrollierte weiße bis der Erwünschte ist Zustand erreicht ist. Deployment können später noch upgedatet, zurückgesetzt, skaliert, gestoppt und wieder fortgesetzt werden.

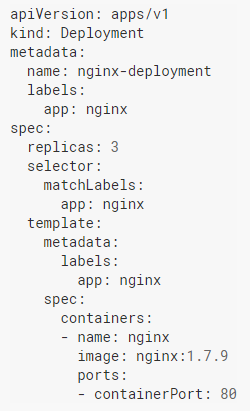
Da das Erstellen von Deployment besonders Wichtig ist, wird hierauf nun noch ein Besonderer Blick anhand eines Beispiels geworfen.

Deployments werden wie schon zuvor erwähnt mit Yaml oder Json Dateien erstellt, im Folgenden handelt es sich um eine Yaml Datei da dies 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.

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.

Abbildung 5: Deplyoment Yaml (Kubernetes)

#### 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.

#### Persistent Volume Claim

Persistent Volume Claim widerspiegeln Anfragen von Nutzern für Speicherplatz. Sowie Pods Ressourcen von Nodes verwenden, verwenden PVCs Ressourcen von Persistent Volumens(PV) oder Storage Classes. Claims werden hierbei immer an eine PV oder SC gebunden wobei beliebig viele Pods PVCs verwenden können.

#### Persistent Volumens

Persistent Volumens sind Speicherplätze im Cluster welche von einem Administrator freigegeben wurden. Sie haben einen unabhängigen Lebenszyklus von den benutzenden Pods und beinhalten die Informationen über die implementierten Speichersysteme.

#### Storage Class

Storage Classes (Manchmal auf Profiles genannt) bieten Administratoren die Möglichkeit unterschiedliche Arten an Klassen/Arten an Speicher festzulegen, sodass sie von PVC verwendet werden können.

### Kubernetes Architektur

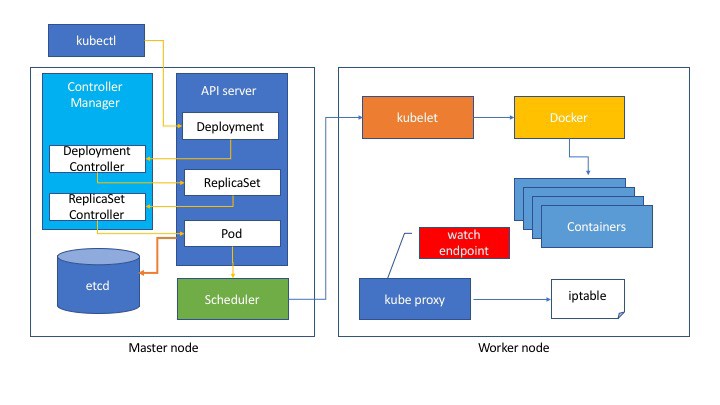


Abbildung 6: Kubernetes Architektur Übersicht (Ushio 2018)

#### Scheduler

Der Scheduler entscheidet auf welchen Nodes die neu erstellten Pods ausgeführt werden. Es können bei Bedarf eigene Scheduler verwendet werden, falls der Entscheidungsprozess nicht den eigenen Wünschen entspricht.

#### Kubelet

Der API Server spricht mit dem Kubelet in einem Node, das Kubelet stellt die Pods bereit welche es durch PodSpecs(YAML oder JSON Dateien) erhalten hat. Abgesehen von den durch Kubernetes erstellten Containern werden nicht vom Kubelet gemanagt. 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 Traffic Regeln in IP Tabellen fest, leitet Traffic um und filtert das Netzwerk.

#### API Server

Der API Server Validiert und Konfiguriert Daten für alle API Objekte wie Pods, Replica Controller, Services, Volumens, Namespaces ... Durch ihn findet jegliche Interaktion über die Kubectl und damit dem Entwickler statt sowie die Kommunikation der anderen Komponenten welche mit dem Ectd sprechen möchten.

#### Controller Manager

Der Controller Manager ist ein Zusammenschluss aller Controller welche in einem einzigen Prozess laufen, die Controller sind jedoch alle Logisch getrennt und gehen eigenen aufgaben nach.

#### Etcd

Etcd ist ein Konsistenter hochverfügbarer Key-Value Speicher, der als Backupspeicher von Kubernetes für alle Clusterdaten verwendet wird. Es kann auch ein Cluster an Etcd angelegt werden falls es mehrere Kubernetes Masters gibt.

#### Container Network Interface

Das CNI ist ein allgemeines Netzwerk welches von den Anwendungen verwendet wird, um Clustern Interne Kommunikation durchzuführen. Die Umsetzung hängt vom Anbieter des CNI ab siehe:

<https://landscape.cncf.io/category=cloud-native-network&format=card-mode&grouping=category>.

## Vereinheitlichte Systeme

### Service-Mesh

#### Warum Service-Meshes entstanden sind?

Ein Service-Mesh ist eine Infrastruktur für ein Netzwerk an Microservices. Infrastruktur bedeutet, dass es keine fachlichen, sondern nur technische Funktionen ersetzt bzw. bietet. Im Gegensatz zu anderen Infrastrukturen wie z.B. Kubernetes steuert ein Service Mesh allerdings auch das Laufzeitverhalten einer Anwendung und nicht nur die Infrastruktur. Durch die wachsende Größe und Komplexität von Microservice Architekturen, wird es immer schwieriger diese zu Überblicken und zu Managen. Es kommen alle möglichen Techniken zum Einsatz, wie schon zuvor in 4.2 Gewährleistung von Services aufgezeigt wurde. Diese sind durchaus auch möglich ohne ein Service-Mesh zusammen zu Benutzen oder sogar Notwendig wenn das Service-Mesh dies nicht zur Verfügung stellt. Es sollten allerdings einige Aspekte zu herkömmlichen Herangehensweisen verglichen werden und Natürlich kommt es auch immer auf die jeweilige Umsetzung des Anbieters an.

#### Was ist/ Was macht ein Service Mesh?

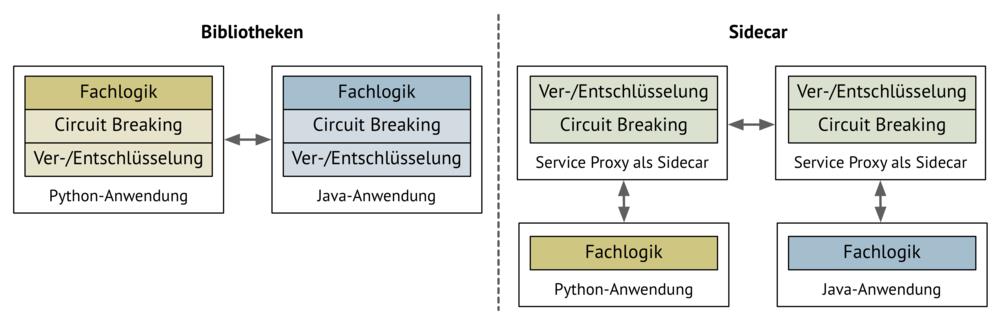
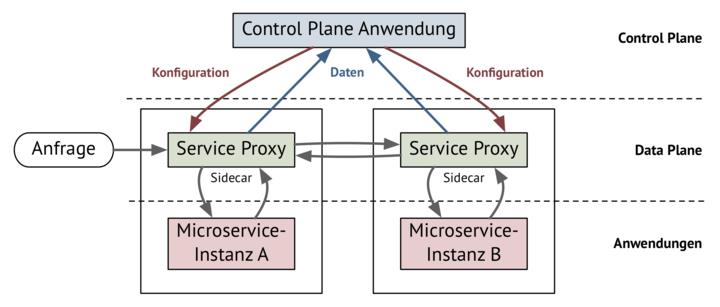
Als Voraussetzung für ein Service Mesh, ist eine Microservice Architektur, da sie genau für diese Umgebung gebaut sind und eine Infrastruktur Komponente wie Kubernetes( Orchestrierung) benutzten können. Diese sind eine allgemeine Voraussetzung für die Service Meshes sind Bsp. Linkerd 2 ist nur mit Kubernetes 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 Beiwagen stammt und auch sehr ähnlich funktioniert. Hierbei werden die Sidecars an bereits bestehende Container geheftet und haben dem entsprechend auch denselben Lebenszyklus. Sidecars übernehmen sprach-unabhängig unterstützende Funktionen für diese Container. Typische Funktionen welche durch Sidecars abgedeckt werden, umschließen Metriken, Routing, Ver- bzw. Entschlüsselung, Service discovery, Health checks, Robustheit 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 21 ist eine Gegenüberstellung zur herkömmlichen Herangehensweise zu sehen mit einer Auslagerung in Sidecars zu sehen.

Abbildung 21: Nutzung von Bibliotheken VS Auslagerung in Sidecars (Prinz 2019)

Alle Services mit ihren Sidecar Proxys werden zusammen als Data Plane bezeichnet und bildet ein von zwei großes Teilen eines Service Meshes. Sidecars sind jedoch keine neue Erfindung, sondern gibt es sie schon eine ganze Weile, was neu ist, ist das sie von einem zentralen Punkt verwaltet und automatisch verteilt werden.

Der Zweite große Teil eines Service Meshes bildet die Control Plane. Die Control Plane bildet das zentrale Gehirn eines Service Meshes, es kommuniziert mit den einzelnen Service Proxys. Das bedeutet die Control Plane verteilt Konfigurationen welche in ihr eingestellt wurden an die Proxys, siehe Abbildung 22 rote Pfeile. Weiterhin sammelt die Control Plane Monitoring Informationen der Proxys ein, siehe Abbildung 22 blaue Pfeile um sie an ein entsprechendes System weiterzugeben wo sie dann Verarbeitet und in z.B. einem Prometeus(eines von möglichen Monitoring Dashboards) Dashboard wiedergeben werden kann. Im Weiteren ist eine Übersicht der Interaktion in der Architektur zu Sehen.

Abbildung 22: Service Mesh Architektur (Prinz 2019)

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

Service Meshes besitzen noch einige Nachteile, welche jeweils abgewogen werden sollten um Herauszufinden, ob es sich lohnt in diese zu Investieren oder eben nich. Im Moment ist ein größerer Nachteil, dass die jeweiligen Technologien relative Neu sind und erst eingeschätzt werden müssen wie sie sich in kleineren oder vor allem in größeren Projekten beweisen. Die nicht unbedeutende Auswahl an unterschiedlichen Anbietern für Service Meshes macht dies nicht gerade einfacher und eine Auswertung der Technologie unübersichtlicher. Man kann sich ein eigenes Bild z. B. über das Service Mesh Landscape von (Layer5 10/28/2019) machen. Zwei der 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 hinzu, welche die Komplexität der Umgebung erhöhen kann und zusätzliche Kosten verursacht, welche auf das Projektbudget drücken.

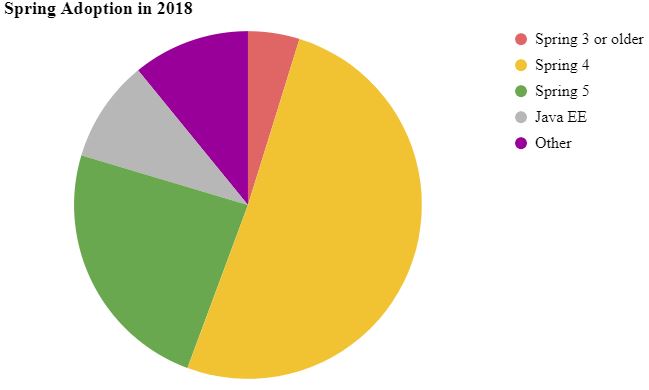
Erhöhte Latenzzeit, welche durch zusätzliche Proxys vor jedem Container entsteht, wodurch speziell die innere Kommunikation durch 2 weitere Hops verlangsamt wird.

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

### Spring Cloud

#### 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 benutzerfreundlich 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 zum bisher weit verbreiteten und zuvor Präsenten Java Enterprise Edition kurz JEE Framework geliefert. Wie in Abbildung 23 zu sehen ist, hat das Spring Framework das zuvor vorherrschende Java EE Framework größtenteils abgelöst.

Abbildung 23: Spring Annahme (Paraschiv 2018)

Da das Spring Framework von Netflix mit entwickelt wurde, welche ebenfalls eines der größten verteilten Systeme weltweit besitzen und es zur damaligen Zeit noch keine etablierten Technologien gab, diese nicht praktikabel oder ausreichend waren um ein robustes System in diesem Ausmaße zu etablieren haben sie hier ihre eigenen Werk-zeuge gebaut. Diese Tools bilden einen wesentlichen Bestandteil dessen, was mit einer herkömmlichen Java Programmierung an Robustheit erzeugt werden kann. Dies beinhaltet eine zum einen Infrastruktur Lösungen wie Service Discovery, Distributed Tracing, verteilte Konfiguration oder Clientseitiges Load balancing. Als auch Funktionen zur Festigung der Robustheit wie Circuit Breaker, Timeouts oder auch Retry Regeln. Fast jede dieser Komponenten, kann im Java Code spezifisch konfiguriert/Code außen herum geschrieben werden und bietet dadurch deutlich mehr Möglichkeiten für Entwickler mit entsprechenden Problemen umzugehen um z. B. Fallbacks im Circuit Breaker einzustellen. Da nie das Java Umfeld 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 täten.

# Möglichkeiten zum Management und Gewährleisten eines verteilten Systems

## Kommunikations-Leitung

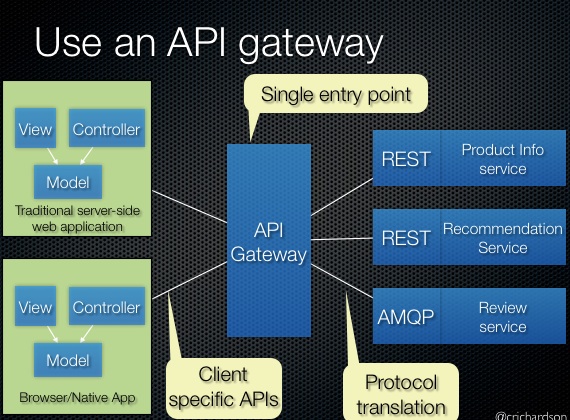
### API Gateway

#### Wieso sollten API Gateways benutzt werden?

Da wir in einer Microservice Architektur unsere Informationen von vielen unterschiedlichen Punkten beziehen, entsteht die Frage wie die Nutzer unserer Anwendung die einzelnen Services erreichen können bzw. sich diese im Hintergrund ändern können. Und dies ohne großen zusätzlichen Konfigurationsaufwand. Außerdem sollten Services Protokoll unabhängig kommunizieren können, um Entwicklern größere Freiheiten zu ermöglichen bzw. für Performance zu Optimieren. Durch die Aufteilung von Services in unterschiedliche Bereiche kann es unübersichtlich werden, wenn diese im selben Netzwerkraum verbleiben.

#### Was ein API Gateway erreichen soll

* Services können unterschiedliche, auch Web inkompatible, Protokolle nutzen siehe Abbildung 7.
* Clients benötigen Daten von vielen unterschiedlichen Quellen/Microservices, was dem Client allerdings nicht bekannt sein soll (keine Informationen über internen Datenverkehr) siehe Abbildung 7 rechts.
* Eine Client entsprechende Daten Bereitstellung sollte Erreicht werden (Desktop vs. Mobile) siehe Abbildung 7 links.
* Services können sich sowohl in Größe als auch Ort ändern und sollen gleichzeitig vom Clients versteckt 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 7 im Gateway).

Abbildung 7: API Gateway Übersicht (Richardson 11/20/2019)

#### Ein Gateway als Lösung

Der Gateway dient als einziger eingangs Punkt für alle Nutzer, ist dies nicht der Fall können Nachrichten unbemerkt an die APIs geladen. Hierbei übernimmt der 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. Der API Gateway kann jedem Nutzer eine andere API zur Verfügung stellen und dadurch die entstehenden Bedürfnisse spezifischer handhaben. Es ist auch, Entsprechend von welchem Anbieter der Gateway stammt, möglich Autorisierung, Authentisieren, Logging, Monitoring usw. in dem Gateway zu verschieben/Aktivieren und somit den Client noch dünner zu Gestalten bzw. Dritt Software nicht zu benötigen.

#### Backends for Frontends

Einen etwas anderen Ansatz nimmt die Variante welche „Backends for Frontend“ genannt wird. Für jedes Frontend (sowohl Web Client als auch Mobile oder 3rd Party) wird ein eigener Gateway erstellt welcher auch nur für dieses verantwortlich ist siehe Abbildung 8. Das hat zum Vorteil, dass man zum einen keinen alleinigen Flaschenhals mehr hat, zum anderen das die Gateways zweckmäßig getrennt 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.

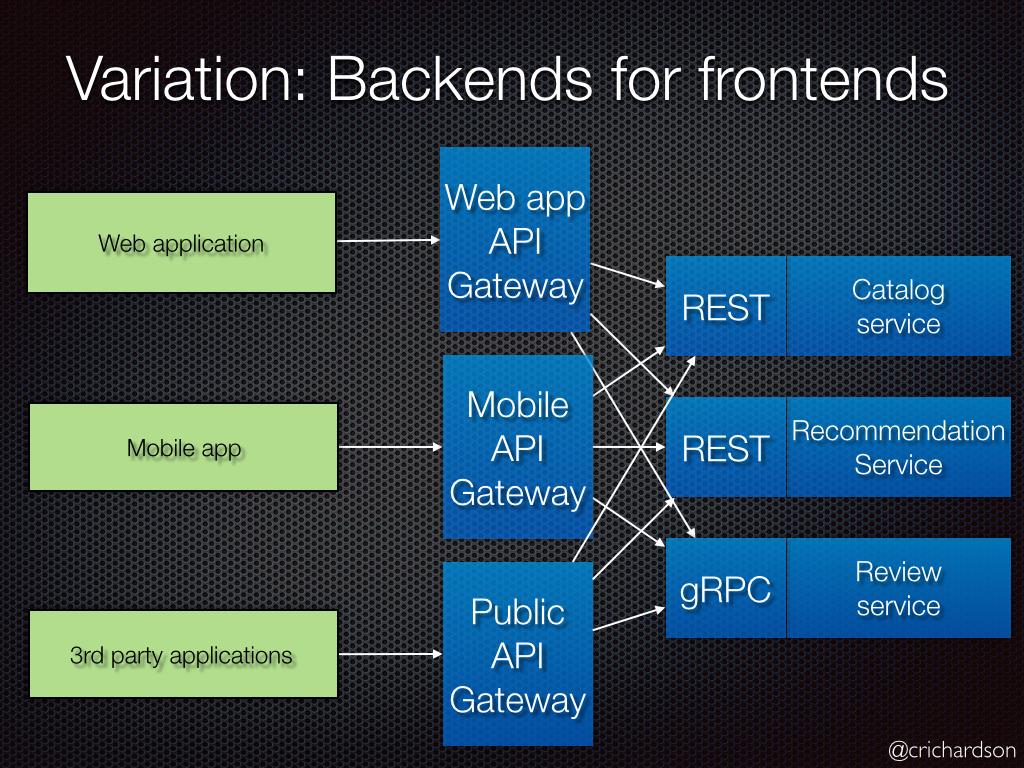


Abbildung 8: API Gateways Backends for frontends (Richardson 11/20/2019)

### API Microgateway

Der API Microgateway, ist üblicherweise ein Proxy (Kommunikationsschnittstelle), entweder extern oder als sidecar, welcher vor einem Microservice sitzt. Er 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 der Gateway, micro also klein genug bleibt und nicht zu groß und schwer wird können beliebige Funktionalitäten hinzugefügt werden. Man sollte immer bedenken, dass der Microgateway in den Häufigsten fällen im selben Bereich wie der Service läuft und infolgedessen, dessen Startzeit mit beeinträchtigt. Doch durch diese enge Bauweise wird er auch sehr wiederverwendbar, welche 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 Sicherheit Informationen über z. B. API Services. Der API Microgateway wird inzwischen vor allem in Service Meshes wie zuvor Erklärt verwendet und findet dadurch eine große Anwendungsfläche.

## Gewährleistung von Services

### Einleitung

Microservices, ohne robuste Systeme, sind wie ein Mann mit Krücken welcher versucht zu rennen. Er wird unweigerlich Fallen, vor allem wenn die Umgebung Schwieriger wird. Da Microservices für Skalierbarkeit gebaut werden, werden sie in einem Laufenden System immer Mehr werden, davon ausgegangen das dass Produkt auch Erfolgreich ist. Mit einer hohen Anzahl an Microservices kommen auch unweigerlich einige Probleme mit sich, um diesen Vorbeugend entgegenzuwirken benötigen wir einige Robuste Systeme welche im Folgenden genauer betrachtet werden und auch genauer auf die Spezifischen Probleme eingegangen wird.

### Bulkhead

#### Microservices Probleme ohne Bulkheads

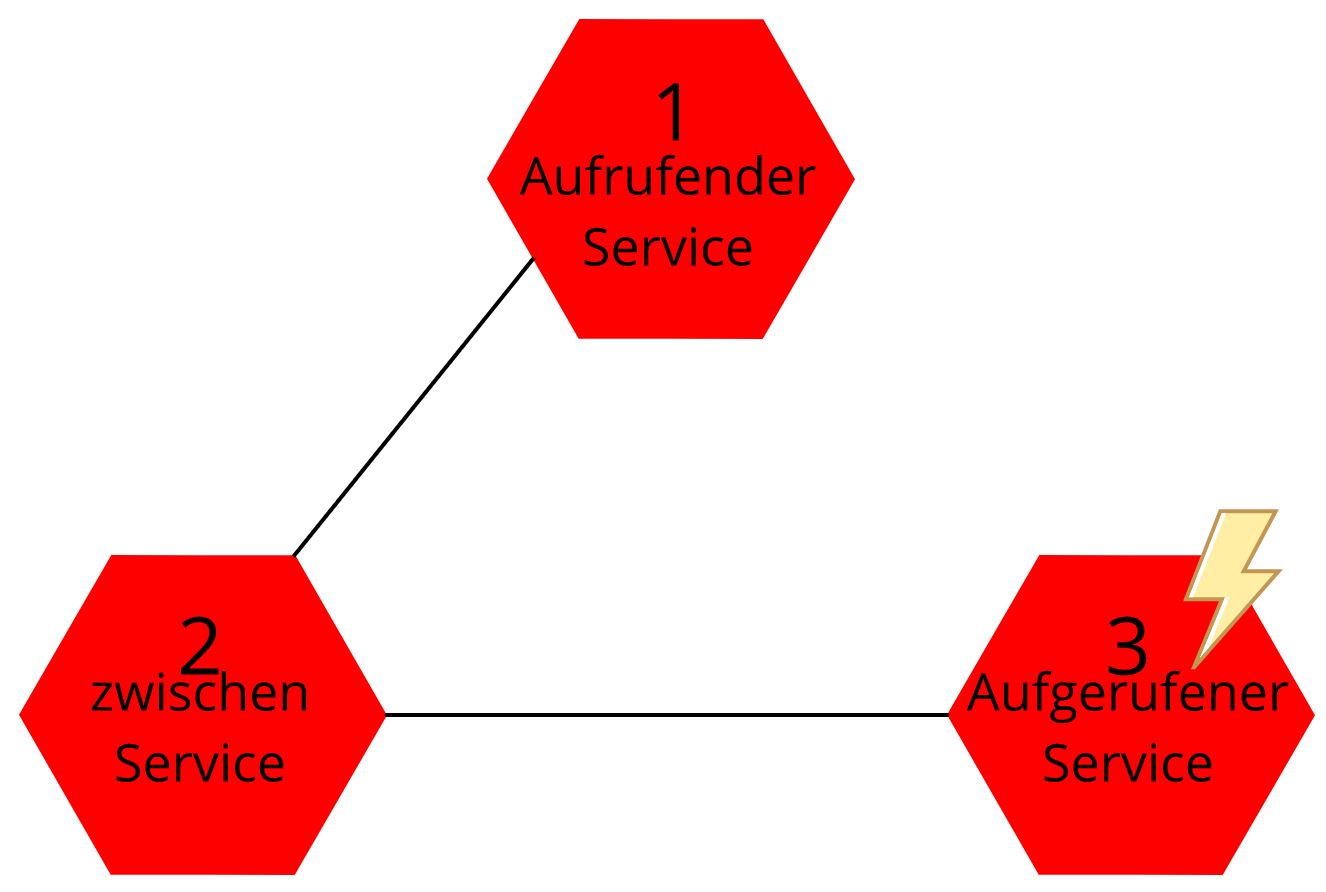
Wenn in einem Softwaresystem ein Fehler verursacht wird stürzt in der Regel das gesamte System ab. In einem verteilten System heißt dies, dass wenn ein Fehler in einem Teilsystem verursacht wird, alle verbundenen oder gar das gesamte System abstürzt siehe Abbildung 9. Weitere Probleme welche in einem verteilten System auftreten können betreffen die Ressourcen.

Abbildung 9: Microservices ohne den Bulkhead (Eigenkreation)

So kann ein einzelner Service Ressourcen von mehreren Services aufbrauchen bis diese gar nicht mehr arbeiten können und im Endeffekt mit abstürzen/ausfallen zwei Beispiele hierfür wären: zwei Services besitzen dieselbe Warteschlangenliste, der erste Service schreibt sie voll woraufhin der zweite Service sie auch nicht mehr verwenden kann oder Es entsteht ein Fehler in einem Service B, Service A der diesen Service anspricht macht nun ganz schnelle und häufige anfragen da immer sofort eine Antwort mit einem Error zurück gesendet wird wodurch auch Service A ausfällt da dieser hierdurch überlastet wird.

#### Was sind Bulkheads?

Bulkhead ist ein Begriff, welcher aus dem Schiffsbau kommt und beschreibt eine Technik, bei der das Schiff in Segmente unterteilt wird. Bei einem Leck können diese Segmente separat geschlossen werden, um somit zu verhindern, dass das gesamte Schiff voll Wasser läuft siehe Abbildung 10 und Abbildung 11. Feuerschutztüren im Brandschutz dienen demselben Zweck und verhindern die Rauchverbreitung im Haus.

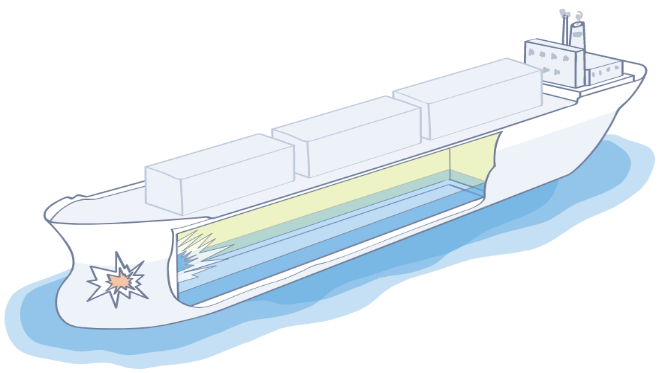
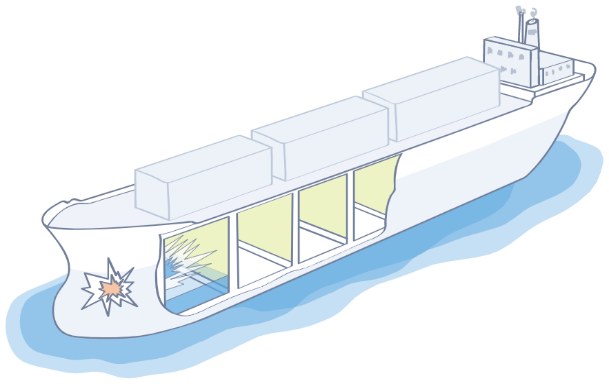
Abbildung 10: Schiff Links Ohne Bulkhead (IBM 11/28/2019)

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

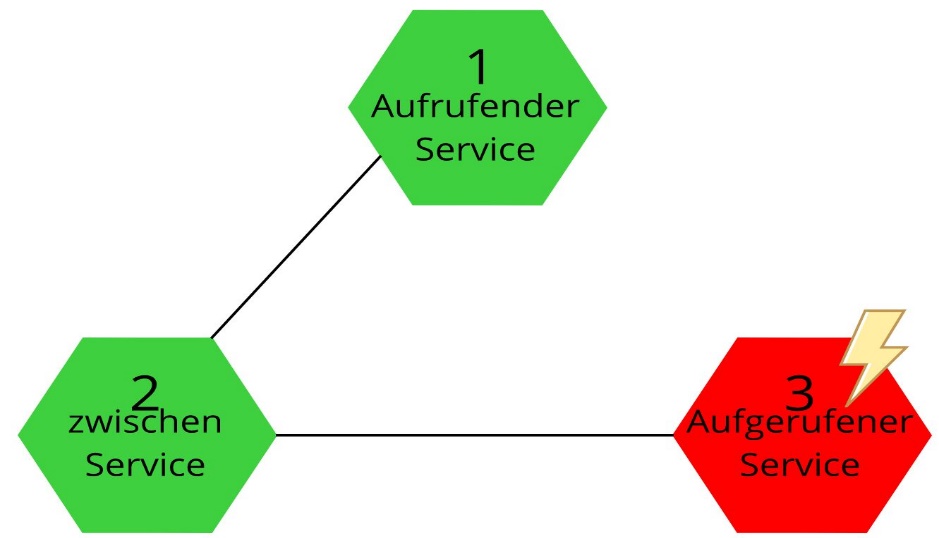
So wie im Schiffsbau und im Brandschutz, wird das Bulkhead Pattern (in Deutsch Schott Muster) auch in der Microservice Architektur verwendet, um zu verhindern, dass das gesamte System durch einen einzigen Fehler zum Einsturz gebracht wird. Dies geschieht im Falle von Microservices auf die Art und Weise, 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 werden.

Durch dieses Pattern wird einerseits zwar der Schaden nicht verhindert dafür allerdings davon abgehalten sich auszubreiten und in einem Rahmen zu halten wo sich ein Service auch wieder erholen kann. Andererseits dient es dazu nur jeweils so viel Arbeit auf einmal abzuarbeiten wie dies Eingestellt ist, um immer noch reagieren zu können. Wenn man beim Bild des Schiffen bleibt, wiederspiegeln die Kammern jeweils einen Service mit einer gewissen Anzahl an Ressourcen und dieser Service kann nur so viel Einnehmen/Fluten wie seine Kammer hergibt/bereitstellt.

#### 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. teilen. Durch Kostenmanagement kann man es auch nur auf die Services herunterbrachten.

Abbildung 12: Microservices mit dem Bulkhead (Eigenkreation)

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 werden und somit den Schutz welchen Bulkheads gewähren Verletzten.

#### Umsetzung

Es gibt verschiedene Möglichkeiten ein 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 durch Firewall regeln oder z. B. durch Konfiguration in einem Service Mesh. Die **Komplexität** der Implementierung meint hierbei wie spezifisch auf das Geschehen des Prozesses Einfluss genommen werden kann. Diese beiden Eigenschaften sind oft zwei Seiten einer Wage, falls es bevorzugt ist eine Implementierung direkte im Code durchzuführen so kann man Automatisch auch spezifischere Lösungen Produzieren. Ist ein Ansatz in einem höheren Level gewünscht, wodurch auch weniger Code selbst geschrieben werden muss, so kann oft auf bereits bestehenden Funktionen zurückgegriffen werden. Durch die Verwendung von Funktionen in höheren Ebenen wird die Umsetzung allerdings auch automatisch limitiert, so können teilweise nur Funktionen wie Ratelimiting eingebunden werden. Da kein direkter Zugriff auf Prozessinterne Ressourcen besteht, stellt sich hierbei auch die Frage, dass wenn eine Anwendung nicht in unterschiedliche Ressourcen abschnitte getrennt werden kann, überhaupt von einem Bulkhead die Rede ist.

Beispiel: Eine Code Umsetzung in einfachem schlichtem Java mit dem Resilient4j.

Wie man 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 spezifische 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 erzwungen wird welche in einer Microservice Umgebung bereits Standard seien sollte. Falls man die Synchrone Semaphore Version des Bulkheads verwendet fallen allerdings auch einige Einstellungsmöglichkeiten weg, wodurch man nicht einen so genauen Einfluss nehmen 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 als zuvor im Reihnen Javacode man in Codeteil 3 sehen kann. Die alleinige Voraussetzung zur Nutzung des Threadpool Bulkheads anstelle des Semaphoren Bulkheads ist, die das ein CompletableFutur Objekt von der Methode zurückgegeben wird. Einstellungen werden in einer application.properties oder .yml Datei festgelegt und bieten viel Spielraum zum Einstellen der Funktionalität. Falls man mehrere resilient Funktionalitäten verbinden möchte, geschieht dies von Haus aus in folgender Reihenfolge:

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

Resilient4j ist hierbei die am weitesten verbreitete Bibliotheken im Java Umfeld und Löst damit denn bisherigen Platzhirsch Hystrix von Netzflix ab, welcher nicht länger supportet wird

Beispiel: Bulkhead durch Firewall Richtlinien in Kubernetes

Beispiel: Bulkhead durch Konfiguration in Istio

### Circuit breaker

#### Kurzschlüsse in Softwaresystemen

Kurzschlüsse geschehen, wenn zu viel Last/Strom auf einer Leitung ist und diese dann durchbrennen und Feuer verursachen können. In einem Softwaresystem sieht das so aus, dass so viel Last an einem Service anliegt das dieser Abstürzt und sobald er wieder neugestartet wird, in seiner Startphase erneut abstürzt da er noch gar nicht für seine Arbeit bereit ist.

#### Was sind Circuit breaker?

Circuit breaker, zu Deutsch Sicherung, kommen ursprünglich aus dem elektrischen Bereich. Sicherungen sind kleine Drähte oder Widerstände welche bei einer gewissen Leistung durchbrennen, bevor Hauptleitungen z. B. in der Wand durchbrennen und so ein Feuer auslösen können. Dies hat ursprünglich öfters zu Hausbränden geführt. Genauso werden sie als Notschalter benutzt, um größere Schäden zu verhindern.

Dieses Prinzip, würde nun anhand eines Programmiermusters in der Software übernommen, umso eine erhöhte Stabilität zu gewehrleisten. Wie in der herkömmlichen Art und Weiße, gibt es auch hier mehrere zustände des „Schaltkreises“. Er kann Offen, geschlossen oder Zusätzlich halb-offen sein. Die Einzelnen Zustände werden nun anhand des Folgenden Bildes (siehe Abbildung 13) erklärt.

**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.

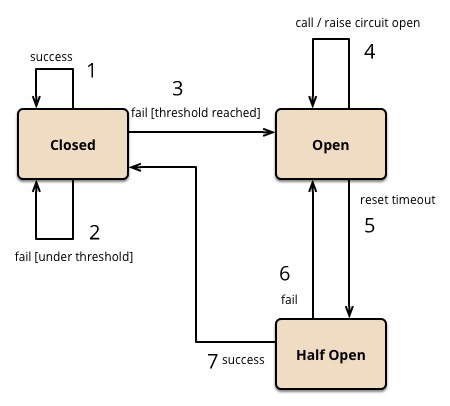
**Open**, blockt erst einmal jeglichen Datenverkehr [4] und schaltet erst nach einer gewissen Zeit auf Half-Open um [5]. Hierbei werden sofort Fehler Meldungen zurückgesendet. In dieser Zeit soll sich der Service, von dem ankommenden Datenverkehr erholen können oder die Zeit haben neu zu starten. Die Wiederherstellung der Verbindung wird in der Regel von externen System übernommen und findet nicht im Circuit breaker statt.

Abbildung 13: Circuit Breaker Zustände (Fowler 2014)

**Half-Open** testet den Service. Es werden hierbei ein paar Anfragen angenommen. Die Anzahl ist normalerweise geringer 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 in Zwei unterschiedliche Arten von Circuit Breakern unterscheiden. Diese unterscheiden sich je nachdem in Managementaufwand und möglicher Einflussname, welche mit erhöhtem Programmieraufwand Hand in Hand geht.

#### Zentral Verwaltete Circuit breaker

Beschreibt einen von einer Middleware verwalteten Circuit breaker. Normalerweise ist dies ein API Gateway, ein Service Mesh oder ein Reverse Proxy. In diesen Fällen geht der gesamte Datenverkehr durch diese Middleware und wird 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 das 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 man auf diese Art und weiße keine einzelnen großen Schwachstellen mehr hat, 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 hinzu kommt, was vor allem der Fall ist wenn unterschiedliche Service in verschiedenen Programmiersprachen geschrieben werden oder Entwickler noch neu in dem Bereich der weiterführenden Fehler sind. Es gibt allerdings in allerlei Sprachen, gut geschriebene Bibliotheken welche weit verbreitet sind und Benutzt werden, was dabei hilft diese Probleme zu bändigen.

#### Umsetzung

Da Circuit breaker mit Nachrichten arbeiten, können diese auf fast allen gängigen Ebenen eingesetzt werden. Hierbei unterscheiden sie sich dadurch wo sie ausgeführt werden, wie spezifisch diese Eingestellt werden können und wie einfach die Verwendung ist. Es kann schlau sein einen Circuit breaker möglichst früh im eigenen Netzwerk einzubauen um darunterliegende Systeme zu schonen.

Beispiel: 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 CircuitBreaker Annotation und entsprechenden Namen versehen werden siehe Codeteil 6. 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 wiederspiegelt.

@GetMapping(path = **"resilience"**)  
@CircuitBreaker(name=**"BACKEND"**)  
**public** CompletableFuture<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

Falls ein Dashboard zur Übersicht der vorgehenden Prozesse gewünscht ist hierbei allerdings etwas mehr notwendig als bei der älteren Alternative Hystrix.

Alternative nur mit Service Mesh wie Istio

### Retry

#### Das zu Lösende Problem

Wenn Anwendungen über das Netzwerk miteinander kommunizieren, kann es immer wieder vorkommen, dass gewisse Fehler auftreten. Anfragen können 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 steht nun also fest, das Fehler immer wieder auftreten und das wir mit ihnen umzugehen haben. Wenn unsere Anwendung also feststellt, dass ein Fehler aufgetreten ist, als sie versucht hat einen Aufruf auszuführen haben wir eine Reihe an Möglichkeiten mit diesem umzugehen:

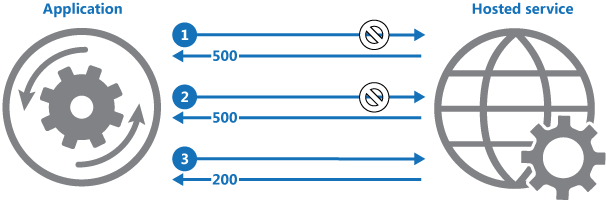
* Abbruch: Falls erkennbar ist das der Fehler sich nicht mit einer Wartezeit beheben lässt oder mit dem Kommunikationsweg zu tun hat, sollte die Aktion abgebrochen werden und eine Exception ausgelöst werden. Wenn Beispielsweise ein Login mehrfach mit den Falschen Daten ausgeführt wird, macht es keinen Sinn diesen überhaupt abzusenden.
* Retry: Wenn der Fehler nur selten vorkommt und keine Rückschlüsse auf konkrete Fehler aufweist, sollte dieser einfach sofort wiederholt werden, da hier mit erhöhter Wahrscheinlichkeit das Netzwerk die Tatsächliche Fehlerquelle ist, was bedeutet das der Fehler nicht nochmal auftreten wird Bsp. Abbildung 14.

Abbildung 14: Retry Pattern (Microsoft)

* Retry mit Verzögerung: Falls es sich um einen eher gewöhnlicherer Fehler handelt, z. B. der Angefragte Service oder das Netzwerk sind Überlaufen. Ist die beste Option die Anfrage mit ein gewissen Verzögerung erneut zu Senden.

#### Umsetzung

Die Retry Funktion von Resilient4j in Verbindung mit Spring Boot 2 ist sehr einfach einzubinden, bietet auch einiges an zusätzlichen Einstellungen, welche bei anderen alternativen so nicht vorhanden sind. (Spring.Rety&&HystrixRetry Nachschauen!!)

@Retry(name=**"BACKEND"**,fallbackMethod = **"secondFallback"**)  
@GetMapping(path = **"resilience"**)  
**public** CompletableFuture<ModelAndView> getItems()  
{

Codeteil 9: Method mit Resilient4j Retry

### Timeout?

### Rate Limiting

Rate Limiting Komponenten sind Circuit Breakern ziemlich ähnlich, in der Weise das 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 auch nicht eine 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 sie umso Wichtigere für kleinere Anwendungen und man sollte sich vermehrt überlegen sie Einzubauen.

Abbildung 14: Rate Limiter (Storozhuk 2018)

Sie 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 14 die blauen Flächen werden nach hinten verschoben und später verarbeitet. Nicht nur kann man nach unterschiedlichen Anfragen Typen filtern und diese Separat handhaben, kategorisieren und unterschiedliche Limits für die entsprechenden Gruppen geben. 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 was im Teil 4.1 Kommunikations-Leitung beschrieben wurde.

### Fallback

#### Was Passiert bei Fehlern?

Wenn Services ausfallen ist es das einfachste, einfach einen Fehler für den Nutzer auszuwerfen. Es ist allerdings auch die wohl unschönste Antwort welche einem Nutzer geliefert werden kann und da wir unsere Systeme in der Regel auch für unsere Nutzer bauen, sollten sie auch die höchste Priorität haben.

#### Das Fallback Pattern

Das Fallback Pattern ist in diesem Falle mehr eine Ergänzung zu Pattern wie Retry und Circuit breaker welche gewährleisten das bei Fehlern unsere Services sich erholen können bzw. Anfragen bei Fehlern wiederholt werden. Sorgt das Fallback Pattern dafür, dass bei Auftretenden Fehlern auf mögliche Nützliche Informationen zurückgegriffen wird. Fallbacks führen hierbei nur wenig bis gar keine Verarbeitung durch, da wir uns ja bereits im Fehlerzustand befinden. Hinzukommt das die entsprechenden Fallbacks nur eine geringe Wahrscheinlichkeit haben sollten selbst Fehlzuschlagen.

Im Folgenden drei unterschiedliche Möglichkeiten einen Fallback einzubauen.

* Stumm Fehleschlagen: Ist die einfachste und vermutlich am weitest verbreiteten Art eines Fallbacks. Hierbei wird einfach ein null wert zurückgegeben, was dann nützlich sein kann wenn die angefragte Ressource optional ist und in der Antwort einfach weggelassen werden kann.
* Speziell Fehlschlagen: Wird in Fällen benutzt wenn es Fallback Methoden in einer Client Bibliothek gibt. Alternative können wir versuchen Lokal gecachte Daten zurückgreifen umso eine geeignete Antwort zu generieren.
* Schnelles Fehlschlagen: Wird verwendet wenn die Daten benötigt werden und es keine wirkliche alternative Möglichkeit gibt. In diesem Fall stellen wir das wohl des Services über die der Nutzererfahrung und Erlauben dem Service sich zu erholen, sodass der Normale Verlauf wieder hergestellt werden kann.

#### Umsetzung

**public** CompletableFuture<ModelAndView> secondFallback(CompletionException e) {

**return** CompletableFuture.*completedFuture*(modelAndView);  
}

Siehe Codeteil 2 und Codeteil 7 für eine mögliche Umsetzung.

## Kommunikations- Überwachung und Verfolgung

### API Management

#### Teilen zu unseren Bedingungen

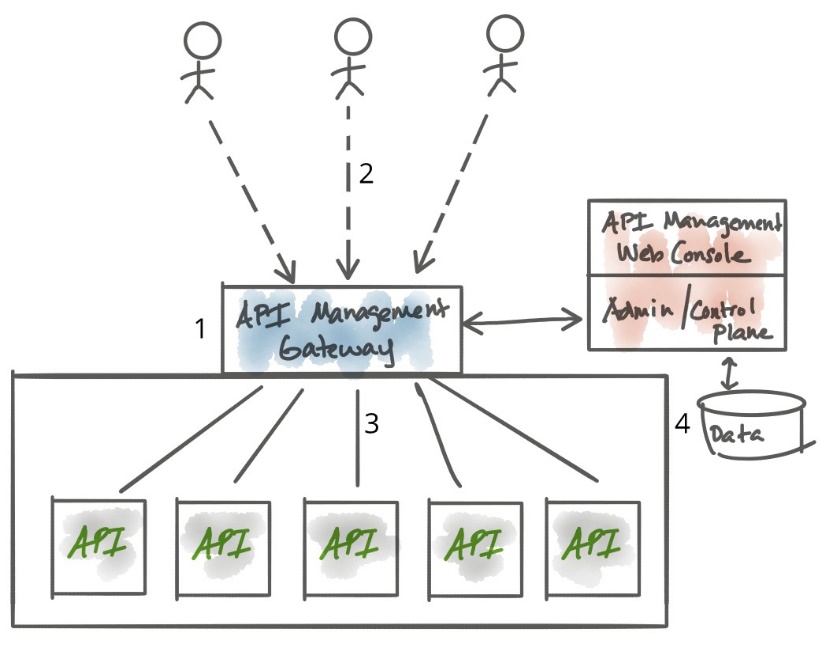
In einem verteilten System können eine Vielzahl an unterschiedlichen APIs vorhanden seien. Diese zu Verwalten und entscheiden zu können wann etwas Öffentlich zugänglich ist oder auch für wenn diese APIs zugänglich sind, wird immer wichtiger.

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)

#### Der API Management Gateway

Der API Management Gateway ist eine Komponente, welche eine weitere Schicht Bildet oder den bisherigen Gateway komplett ersetzt (dies kann auch auf von der Größe des Systems abhängen). Er 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 15 [3] werden von diesem Gateway gemanagt, katalogisiert und verwaltet, wodurch diese vergeben, gefunden und effektive kontrolliert werden können.

Die Managementkomponente kann als Einstiegspunkt in unser System dienen wie es in Abbildung 15 [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 15: API Managment Gateway Overview (Posta 2019)

Ein Punkt bei dem hierbei aufgepasst werden muss, ist keine Businesslogik in diese Schicht Abbildung 15 [1] einfließen zu lassen. Da das API Management eine geteilte Komponente ist, hat es Tendenzen dazu ein all wissend, all verarbeitendes Konstrukt zu werden, wo jeglicher Datenverkehr für Änderungen an APIs durchfließen muss, was erneut in einem organisatorischen Flaschenhals enden kann. Da auch die Management Komponente Skaliert werden muss sollte hier möglichst versucht werden keine Abhängigkeiten zu anderen Komponenten zu erzeugen und genau aus diesem Grund auch eine eigene lokale Datenbank zur Verfügung zu stellen Abbildung 15 [4].

### Distributed tracing

#### Was ist Distributed tracing

Bei distributed tracing geht es darum Anfragen, über mehrere Systeme hin, zu Verknüpfung. Die Verknüpfung geschieht durch zugehörige eindeutige IDs, welche im http Kopf mitgeliefert werden. Die IDs werden in Zusammenhang gebracht umso denn gesamten weg einer Anfrage nachvollziehen zu können. Das heißt es ist leichter möglich herauszufinden wo ein Fehler tatsächlich entstanden ist und welche Systemkomponente verantwortlich ist.

#### Wieso ist Distributed tracing wichtig

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 damals noch lokal gedebugt werden konnte, verteilt sich nun über das Netzwerk hinweg über etliche Hops.



Abbildung 16: Distributed Tracing (Kyma)

Da niemand gerne viel Zeit damit verbringt sich durch etliche Netzwerklogs durchzuwühlen wird eine automatisierte Lösung benötigt. Um genau dieses Problem zu beherrschen 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 Groß werden und sind fast nicht komprimierbar was vor allem an ihrer notwendigen Einzigartigkeit liegt und je nach Systemgröße sehr groß/viele werden können „Tracing is high-volume and high-cardinality“ (McAllister 2019, 6:5-6:10). Es gibt einige „Standards“ (anstatt einen)was dazu führen kann, dass es einen Anbieterwechsel besonders schwer macht, "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).

## Service discovery

### Wo sind alle Services?

Mit der Microservice Architektur, kommen anstelle von ein paar wenigen schweren übersichtlichen bzw. fest zusammenhängenden Services, unglaublich viele, schlanke, kurzlebige und autoskalierte Services zum Einsatz (Abbildung 10: Why to use Service discovery (Richardson, Chris of Eventuate, Inc. 2015)). Und all diese neuen Services sollen nun auch miteinander kommunizieren können. Wo es bisher noch möglich war Manuel Services zuzuweisen, Adressen Festzulegen und diese zu verbreiten, ist nun vollkommen Unvorstellbar und muss automatisiert werden.

Abbildung 17: Wiso 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 einmal anmelden, deshalb wird diese Komponente auch Service Registries genannt. Im selben Schritt wird in diesem Prozess auch eine Lastenverteilung durchgeführt wodurch sie nichtmehr 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 die 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).

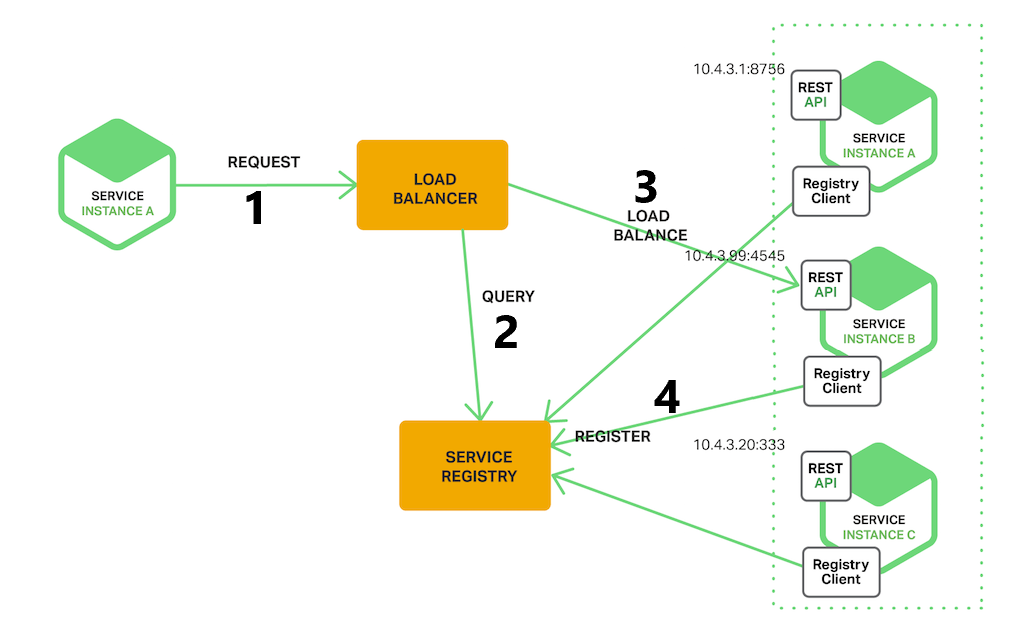


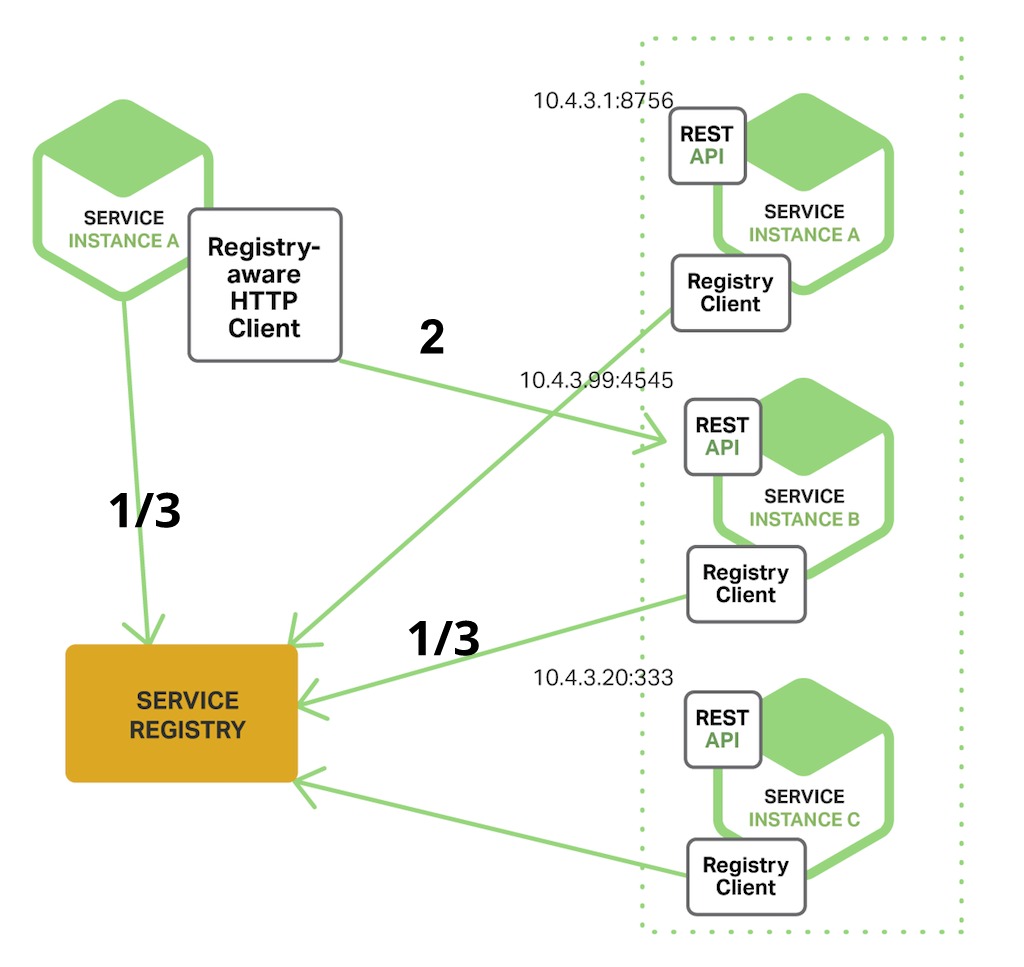
Abbildung 18: Server-Side discovery Pattern (Richardson 2015)

Auf die genauen Registrierungsverfahren [4] wird später eingegangen.

Wenn Serverseitige discovery benutzt wird kommen einige Vor- und Nachteile mit sich. Durch die Abstraktion des discovery Mechanismus, ist es möglich Clients/Services unabhängig von ihrer Sprache zu benutzten und entfernt die Notwendigkeit diese zu Implementieren. Außerdem stellen einige Bereitstellungsumgebungen(Kubernetes) diese bereits frei zur Verfügung. Das Problem dieser Technik ist, dass durch die Einfügung 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 19: 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.

### Selbstregistrierungs Methode

Jede Service Instanz ist in dieser Methode, selbst dafür verantwortlich sich an der Registry Anzumelden (Abbildung 12: Client-side discovery Pattern (Richardson, Chris of Eventuate, Inc. 2015)[3]). Außerdem sendet der Client die Herzschläge(kontinuierliche Abfragen) anfragen um nicht abgemeldet zu werden. Die Nachteile gleichen denn der Clientseitigen Methode, der Registrierungscode muss für jede Sprache oder jedes Framework erneut geschrieben werden wenn diese nicht schon vorhanden sind.

#### Umsetzung

Anhand des Service Discovery Servers Eureka wird im Folgenden ein Beispiel vorgezeigt wie eine Clientseitige Discovery stattfindet.

Der Discovery Server:

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

### Drittparteiregistrierungs Methode

Mit 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 Herzschläge oder durch Überwachung der Events des Services (Abbildung 12: Server-Side discovery Pattern (Richardson, Chris of Eventuate, Inc. 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 das 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 der, dass eine weitere Hoch verfügbare Komponente hinzukommt welche gemanagte werden muss.

## Status und Daten Verbreitung in verteilten Systemen

### Daten Konsistenz

Daten Konsistenz ist ein weiteres großes Thema im Bereich der verteilten Systeme und der Kommunikation in diesem Bereich, durch zeitliche und räumliche Eingrenzungen kann in dieses Thema jedoch nicht weiter in die Tiefe gegangen werden und wird dritten überlassen.

### Konfiguration Server

# Umsetzung eines robusten verteilten Systems mit Hilfe eines Service-Meshes

## Das Service-Mesh Istio

Istio ist ein Service-Mesh, es wird hier nicht noch einmal darauf eingegangen was ein Service-Mesh ist, da dies bereits im Punkt 4.9 Service-Mesh erläutert wurde. Istio ist zuallererst Vollständig **Open-Source** und bringt indem Zuge alle seine Vor und Nachteile mit sich. Durch die Benutzung von Istio ist es möglich services zu Veröffentlichen und sie mit unterschiedlichsten Techniken auszustatten(welche Techniken dies genau beinhaltet wird später erläutert) welche mit wenigen bis keinen Code änderungen in Kraft treten können

### Istio Architecture

### Istio Notizen

Wie ruft man andere Services auf?

Wie funktioniert das load-balancing?

Wie kann man äußere Services ansprechen?

Wie funktioniert der Circuit breaker in Istio?

Rate limiting, …

Wie werden side cars Injiziert?

# Zusammenfassung und Ausblick

Zwei Seiten

# Glossar

# Ehrenwörtliche Erklärung

|  |  |  |  |
| --- | --- | --- | --- |
| Name: |  | Vorname: |  |
| Matrikel-Nr.: |  | Studiengang: |  |

Hiermit versichere ich, <Vorname, Name>, dass ich die vorliegende <Bachelor- oder Masterarbeit> mit dem Titel <Titel der Abschlussarbeit> 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.

Ort, Datum Unterschrift

# Stichwortverzeichnis

Abbildung 25

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: 04.10.2019 – 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: 02.05.2017 – 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 02.12.2019

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 07.11.2019

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 08.11.2019

Microsoft, dragon119: *Wiederholungsmuster - Cloud Design Patterns.* URL https://docs.microsoft.com/de-de/azure/architecture/patterns/retry – Überprüfungsdatum 03.12.2019

Paraschiv, Eugen: *The State of Java in 2018.* URL https://www.baeldung.com/java-in-2018#spring-adoption. – Aktualisierungsdatum: 06.11.2018 – 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: 16.10.2019 – Überprüfungsdatum 03.12.2019

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 05.12.2019

Richardson, Chris: *Microservices Pattern: API gateway pattern.* URL https://microservices.io/patterns/apigateway.html. – Aktualisierungsdatum: 20.11.2019 – Überprüfungsdatum 02.12.2019

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 09.10.2019

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

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: 02.05.2018 – 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