Möglichkeiten zur Kontrolle und Steuerung der Kommunikation von Microservices

**Bachelorarbeit**

im Studiengang  
Softwaretechnik und Medieninformatik

vorgelegt von

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

am 1. Januar 2027   
an der Hochschule Esslingen

Erstprüfer/in:   
Zweitprüfer/in:

# Kurzfassung

# Abstract

**Keywords:**

# Inhaltsverzeichnis

Kurzfassung 2

Abstract 2

Inhaltsverzeichnis 3

Abbildungsverzeichnis 5

Tabellenverzeichnis 5

Codeverzeichnis 5

Abkürzungsverzeichnis 6

1 Überblick/Einleitung 7

2 Microservices 8

2.1 Monolith oder Microservices 8

2.1.1 Monolith 8

2.1.2 Microservices 9

3 Verteilte Systeme 12

3.1 Definition 12

3.2 Orchestrierung 12

3.3 Netzwerk/Kommunikation in einem Verteilten System 12

3.4 Von Fehler Potential zu Robustheit 13

3.5 Daten Beständigkeit 13

4 Werkezuge oder Techniken für ein robustes verteiltes System 14

4.1 Service discovery 14

4.1.1 Was ist Service discovery und warum ist sie unabdingbar 14

4.1.2 Serverseitige discovery 15

4.1.3 Clientseitige discovery 15

4.1.4 Selbstregistrierungs Methode 16

4.1.5 Drittparteiregistrierungs Methode 16

4.2 Bulkhead 16

4.2.1 Probleme welche durch das Bulkhead Pattern gelöst werden 17

4.2.2 Prinzipien des Bulkhead Patterns 17

4.3 Circuit breaker 17

4.3.1 Zusätzlichen Abschnitt für Fehler Handhabung schreiben? 18

4.3.2 Unterschiedliche Arten/Ebenen von Circuit breakern 18

4.3.3 Mögliche aktuelle unabhängig Verwaltete circuit breaker 19

4.3.4 Mögliche aktuelle zentral Verwaltete circuit breaker 19

4.4 Retry 20

4.4.1 Das zu Lösende Problem 20

4.4.2 Lösungsansatz für das Problems 20

4.5 Rate Limiting 20

4.6 API Gateway 21

4.6.1 Das zu Lösende Problem 21

4.6.2 Was ein API Gateway erreichen soll 21

4.6.3 Ein Gateway als Lösung 22

4.6.4 Backends for frontends Bzw. Gateways je Nutzeroberflächen 22

4.7 API Microgateway(To Shallow? ) 22

4.8 API Management 23

4.9 Service-Mesh 24

4.9.1 Warum Service-Meshes entstanden sind? 24

4.9.2 Was ist/ Was macht ein Service Mesh? 24

4.9.3 Was Spricht gegen die Nutzung von Service Meshes 25

4.10 Daten Konsistenz???? 25

4.11 Konfiguration Server 25

4.12 Verteilte anfragen Verfolgung/Überwachung (Distributed tracing) 25

4.12.1 Was ist verteilte anfragen Überwachung 25

4.12.2 Wieso ist verteilte anfragen Verfolgung/Überwachung wichtig 26

4.12.3 Nachteile 26

5 Umsetzung eines robusten verteilten Systems mit Hilfe eines Service-Meshes 28

5.1 Das Service-Mesh Istio 28

5.1.1 Istio Architecture 28

5.1.2 Istio Notizen 28

Zusammenfassung und Ausblick 33

Glossar 34

Ehrenwörtliche Erklärung 35

Stichwortverzeichnis 36

6 Literaturverzeichnis 37

# Abbildungsverzeichnis

Abbildung 1 Monolith gegen Microservices Quelle: Eigene Darstellung 8

Abbildung 2 Docker – Beginner’s Guide – Part 1: Images & Containers (Quelle: https://codingthesmartway.com/wp-content/uploads/2019/02/010-1024x500.png) 10

# 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: Beispiel für einen Codeauszug 34](#_Toc419798065)

# Abkürzungsverzeichnis

ALP Arbeits-, Lern- und Präsentationstechniken

HBI Hochschule für Bibliotheks- und Informationswesen

HdM Hochschule der Medien

HSE Hochschule Esslingen

# Überblick/Einleitung

Microservices, sind grob gesehen Teilmodule Vollständer Softwaresysteme und können sowohl physisch, sprachlich und strukturell getrennt sein. Durch diese Trennung, entsteht eine große Menge, als auch unterschiedliche Arten der Kommunikation. Diese Kommunikation bietet viele Möglichkeiten als auch Tücken. Es muss hierbei, wie so oft, darauf geachtet werden diese möglichst zu erkennen und entsprechend zu behandeln.

Die größte Gefahr hierbei ist, dass das System nicht stabiler, sondern womöglich unübersichtlich wird und dem entsprechend schwieriger zu handhaben. Hier wird im Speziellen darauf geschaut, wie wir das System von Microservices, in möglichst jeder Hinsicht robust gestalten können und welche unterschiedlichen Möglichkeiten wir hierbei verwenden können.

Robust bedeutet in diesem Falle, dass bei einem Fehler keine Kettenreaktion stattfinden darf, dass wenn erwünscht Nachrichten immer ankommen müssen, dass wenn ein Service überlastet ist, dieser sich wieder erholen darf, dass Services eigenständig upgedatet werden können, dass auf Leistungsanfragen automatisch reagiert wird oder diese keinen Einsturz ergeben, dass bei Fehlern auf Fallbacks zurückgegriffen wird, wenn dies Business technisch möglich ist, dass kontinuierliche „Healthchecks“(Lebst du noch?) durch-geführt werden, dass Auslastung über die einzelnen gleichen Microservices aufgeteilt wird und dass nachvollzogen werden kann, wie der Traffic verläuft.

Ein weiterer Punkt der Kommunikation, der hierbei untersucht wird, beinhaltet den Kommunikationsfluss welcher mit wachsender Größe sich immer komplizierter gestalten kann.

Bei einer Vielzahl an Microservices, wird es auch Wichtig, dass die einzelnen Services sich finden können bzw. sich an einem zentralen Punkt Anmelden und sichtbar machen können.

Um eine möglichst geschickte Verwaltung von Updates zu Gewehrleisten, sollte man auf keinen Fall, auf einen zentralen Konfigurationspunkt verzichten. Hierdurch können Updates auf Vielerlei Services gleichzeitig geändert werden, ohne irgendein durcheinander zu verursachen.

Dabei werden sowohl Drittanbieter Software, als auch mögliche Software Patterns und Techniken untersucht und in entsprechende Kategorien und Softwareschichten eingeteilt.

Es soll dabei aber nicht um die Organisation oder die Sicherheit dieser Services gehen.

# Microservices

## C:\Users\gwi\AppData\Local\Microsoft\Windows\INetCache\Content.Word\MonolithVSMicro.pngMonolith oder Microservices

Abbildung 1 Monolith gegen Microservices Quelle: Eigene Darstellung

### Monolith

#### Einführung

Ein Monolith ist im Deutschen ein einheitlicher Stein. In unserem Fall widerspiegelt es ein zusammenhängendes Softwaresystem wieder, welches es seit dem Beginn des Internets gibt und alle drei Komponenten in sich Vereinigt. Diese Komponenten sind die UI, die Datenbank und einen Server. Der Code mag in unterschiedliche Teile aufgeteilt sein aber das System wird immer als ganzes, einheitliches Paket veröffentlicht. Bei jeder Änderung muss somit auch das ganze System neu veröffentlicht werden.

#### 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 vor allem bei eingespielten Entwicklerteam offensichtlich wird.
* Durch die zusammenhängende Architektur, gestaltet es sich einfacher, das System auf 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.
* Datenbank ist an einem Punkt wodurch Konsistenz deutlich einfach bzw. überhaupt erreicht werden kann.
* Es ist deutlich einfacher, eine Datenbank konsistent zu halten, da diese eng mit dem Rest des Systems gekoppelt ist.

#### Nachteile der Monolithen

* Je größer die Anwendung wird, also umso mehr Zeilen Code sie enthält und zusätzliche Features implementiert werden umso wertvoller, werden Entwickler die sich schon mit dem ganzen System auskennen. Allerdings wird es genauso auch immer schwieriger für neue Entwickler einzusteigen und eigene Beiträge hin beizusteuern.
* Ein weiteres Problem von zu groß werdenden zusammenhängenden Systemen ist, das jeder Änderung das gesamte System beeinflussen kann. Dies hat zur Folge, dass der Testaufwand bei Änderungen um ein vielfaches größer sein kann. Resultierend kann es schwieriger werden neue Technologien in ein bestehendes System zu integrieren.
* 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.
* Durch eine starke Verzahnung der Teilbereiche können Projekte mit monolithischer Struktur, nicht so leicht viel neues Personal aufnehmen.

### 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 welche einst ein Teilmodul eines Monolithen war. Hierbei sollte klar sein, das Microservices eigentlich immer in Verbindung mit Containern auftreten. Container bilden hierbei ein Gehäuse für eine entsprechende Anwendung mit all ihren Abhängigkeiten, Laufzeitumgebungen, System Werkzeugen, System Bibliotheken und Einstellungen. Die Container bilden Isolierte Bereiche dar mit ihren eigenen Namenräumen, Benutzern und Ressourcen. Sie stellen allerdings keine Vollwertigen Virtuellen Maschinen dar und Besitzen kein eigenes Betriebssystem oder eigenen Kernel, dies ist allerdings auch gleichzeitig der Grund für ihre Schlankere Bauweise (Stephan Augsten 2017).

#### https://codingthesmartway.com/wp-content/uploads/2019/02/010-1024x500.pngVorteile der Microservices

Abbildung 2 Docker – Beginner’s Guide – Part 1: Images & Containers (Quelle: <https://codingthesmartway.com/wp-content/uploads/2019/02/010-1024x500.png>)

* 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. Vorausgesetzt dies ist vom Business Standpunkt auch möglich.
* Die jeweiligen Bereitstellungen, können durch eine Trennung, vom gesamten System, in kleinere Teil Bereitstellungen verändert werden, was es einfacher macht Änderungen durchzuführen. Tests, bleiben ebenso auf kleinere Service Systeme begrenzt, wodurch sie schneller Veröffentlicht werden können.
* Skalieren der Services wird auch einfacher und Effizienter. Im Gegensatz zum Monolithischen System, wo jeweils das volle System hoch skaliert wird, werden im Micoservice 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.
* Services können, durch die Sprachliche Trennung, einfach in anderen Projekten wiederverwendet werden.

#### Nachteile der Microservices

* Es kann schwieriger sein, Fehler in einem verteilten System zu finden als in einem Lokalen zusammenhängenden System. Dies tritt verstärkt auf, wenn die Services von unterschiedlichen Teams entwickelt werden.
* Da jeder Service seine eigene Datenbank besitzen kann, kann dies zu Datensatz Duplikationen führen.
* Die Unterteilung von Services kann zu weit zu gehen und der Überschuss kann die Nützlichkeit überschreiten.
* Die Notwendigkeit, der Verwendung von Docker oder Kubernetes um das verteilte System zu steuern bzw. Bereitzustellen, kann unerwünscht sein.
* Mit vielen Abhängigkeiten zwischen den Services, kann es herausfordernd werden, ein Ende zu Ende Test durchzuführen.
* Kommunikation zwischen Services, kann sehr teuer werden, wenn sie nicht richtig umgesetzt wird und den entsprechenden Anforderungen entspricht.
* Datenbank Anfragen, können dazu führen das zusätzliche Datenbanken angesprochen werden müssen, welche zu anderen Services gehören. Dies kann zu Problemen führen(wenn dies z. B. mehrfach kaskadierend ist).

# Verteilte Systeme

## Definition

Nach (Andrew S. 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. Hierbei wird nochmal in Frontend und Backend Komponenten/Services unterschieden. Ersteres ist dem Nutzer dabei sichtbar wobei Letzteres dies nicht ist, sondern nur intern verwendet wird.

## 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.

(Arnon 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.

# Werkezuge oder Techniken für ein robustes verteiltes System

Werkezuge oder Techniken für ein robustes verteiltes System

Werkezuge oder Techniken für ein robustes verteiltes System

Die Fehlerquellen/Herausforderungen in Detail -> runterschieben in die einzelnen Bereiche.

Dont apply every pattern

Dont decorate everything

Use metrics

Not only Http (not only in the network)

<https://www.youtube.com/watch?v=gvDvOWtPLVY> 20:20

medium.com/@adhorn

## Service discovery

### Was ist Service discovery und warum ist sie unabdingbar

Mit der Microservice Architektur, kommen anstelle von ein paar wenigen schweren übersichtlichen Services, unglaublich viele, schlanke, kurzlebige und autoskalierte Services zum Einsatz. Und all diese neuen Services sollen nun auch noch miteinander kommunizieren. Wo es bisher noch möglich war Manuel Services zuzuweisen, Adressen Festzulegen und diese zu verbreiten, ist nun vollkommen Unvorstellbar und muss automatisiert werden. Hier kommt die Service discovery ins Spiel.

Figure 1: Why Use Service discovery.jpg (Chris Richardson of Eventuate, Inc. 2015)

Sie bietet einen oder mehrere Zentrale Orte wo die jeweiligen gesuchten Services gefunden werden können, wobei sich diese zuerst einmal anmelden müssen, deshalb werden sie 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 Hauptsächlich zwei Herangehensweisen, welche nun angeschaut werden.

### Serverseitige discovery

In dieser Technik wird ein separater Load balancer benutzt, dieser dient auch als Ein-gang für die Clients. Der Load balancer fragt bei der Service Registry nach und leitet die entsprechenden Clients an die gesuchten verfügbaren Instanzen weiter(immer ein zusätzlicher Stopp).

Auf die genauen Registrierungsverfahren 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 diese 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 und erhalten die verfügbaren Instanzen für die jeweilige Aufgabe. Die Clients, benutzen dann einen Load balancing Algorithmus, um einen Service auszuwählen.

Auf die genauen Registrierungsverfahren wird später eingegangen.

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. 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.

### 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. 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.

## 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. Feuerschutztüren im Brandschutz dienen demselben Zweck und verhindern die Rauchverbreitung im Haus.

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 voneinander Technisch getrennt werden.

### Probleme welche durch das Bulkhead Pattern gelöst werden

* Weitergabe von Fehlern: Da das System in einzelne Bereiche Isoliert ist und keine Ressourcen Teilt, können Fehler in anschließenden/verbunden Teilsystemen Isoliert werden. Das heißt wenn ein Service einen Fehler auslöst werden die Restlichen Systeme nicht beeinträchtigt.
* Lärmender Nachbar: Wenn das Pattern richtig umgesetzt wurde, also Netzwerk, Speicherplatz und Rechenleistung getrennt wurden. Stellt dies sicher, dass wenn ein Einzelner Service sehr viele Ressourcen verwendet dies nicht andere Services, außerhalb der Isolierten Zone, beeinträchtigt.
* Ungewöhnliche Bedarf/Nachfragen: Der Bulkhead Schützt Ressourcen in deren Isolierten Zonen davor, dass andere Services ungewöhnliche Anfragen erhalten z. B. wenn viel mehr Nachfragen als sonst stattfinden. Das heißt das nur der Jeweilige Service von TCP Port Auslastung, Datenbank verfall etc. beeinflusst wird.

### 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.

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.

## 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 und 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 [Finale Bild Nummer] erklärt.

**Closed** ist zuallererst der Zustand, welcher den Normalen betrieb widerspiegelt, dieser wird nur verlassen, wenn eine entsprechende Anzahl an Fehlern in einer festgelegten Zeit überschritten wird (threshold). Verlassen bedeutet in den Open Status zu wechseln.

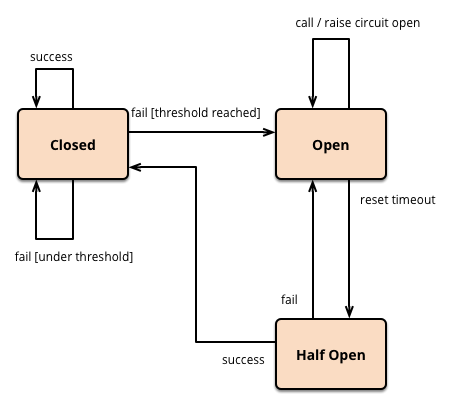
**Open**, blockt erst einmal jeglichen Datenverkehr und schaltet erst nach einer gewissen Zeit auf Half-Open um. Hierbei werden sofort, Error Nachrichten zurückgesendet. In dieser Zeit soll sich der Service, von dem ankommenden Datenverkehr erholen können oder die Zeit haben neu zu starten. Die Wiederherstellung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

Figure 2: Circuit Breaker Zustände (Martin Fowler 2014)

**Half-Open** testet den Service. Es werden 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. Falls die Anfragen weiterhin Fehlschlagen wird wieder zurück in den Open Status gewechselt.

### Zusätzlichen Abschnitt für Fehler Handhabung schreiben?

### 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, die Gefahr ist das dies zu einer einzelnen Schwachstelle, für das gesamte System, werden kann. Der Infrastruktur Architekt sollte also besonders darauf achten das diese Schwachstelle möglichst Ausfallsicher betrieben wird. Das Gute auf der anderen Seite ist, das der Entwickler sich weniger mit diesem zusätzlichen Konstrukt beschäftigen bzw. dieses Entwickeln muss.

#### Unabhängig Verwaltete Circuit breaker

Ein Unabhängig Verwalteter Circuit breaker steht für einen im Service selbst implementierten Circuit breaker welcher den Ankommenden Datenverkehr Regelt. Der große Vorteil ist, das man auf diese Art und weiße keine einzelne große Schwachstelle mehr hat, da diese nach unten auf die Services geschoben wird und ist speziell in hoch Verfügbaren Systemen Wichtig. Der Nachteil ist die erhöhte Komplexität welche für die Entwickler hinzugefügt wird, was vor allem der Fall ist wenn unterschiedliche Service verschiedene Programmiersprachen benutzen oder Entwickler noch neu in dem Bereich der Weiterführenden Fehler sind. Es gibt allerdings in allerlei Sprachen, gut geschriebene Bibliotheken welche weit verbreitet Benutzt werden.

### Mögliche aktuelle unabhängig Verwaltete circuit breaker

resilience4j

### Mögliche aktuelle zentral Verwaltete circuit breaker

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.
* 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.

## 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.



Figure 3: Storozhuk 2018 - Rate Limiter.jpg Load balancer (Storozhuk 2018)

Sie helfen Anfrage spitzen zu verhindern was im Speziellen für kleinere Anwendungen wichtig ist, Anfrage spitzen werden entweder in Warteschlangen abgelegt und später abgearbeitet oder einfach abgelehnt. 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 sie in vielen API Gateways oder Proxys schon direkt mitgeliefert, über diese beiden wird aber später noch gesprochen.

## API Gateway

Wo kann er unabhängig existieren und wo sich mit anderen (API Gateway & Service Mesh) ergänzen und co-existieren.

### Das zu Lösende Problem

Da wir in einer Microservice Architektur unsere Informationen von vielen unterschiedlichen Punkten beziehen müssen, anstatt von einem einzelnen, Entsteht die Frage wie die Nutzer unserer Anwendung die einzelnen Services erreichen können.

### Was ein API Gateway erreichen soll

* Services können sich sowohl in Größe als auch Ort ändern und sollen gleichzeitig vom Clients versteckt sein, also keine direkte Verbindung zu ihm haben.
* Services können unterschiedliche, auch Web inkompatible, Protokolle nutzen.
* Clients benötigen Daten von vielen unterschiedlichen Quellen, welche allerdings nicht den Clients bekannt sein sollen.
* Unterschiedliche Clients benötigen unterschiedliche Mengen/Arten von Daten.
* Es soll auf Hardware Limitierungen, von unterschiedlichen Arten von Clients, Reagiert werden.

### Ein Gateway als Lösung

Der Gateway dient als einziger eingangs Punkt für alle Nutzer. Hierbei übernimmt der Gateway den gesamten Nachrichtenverkehr in beide Richtungen, er wird teilweise nur durchgeschleift zu den entsprechenden Services. Andere Anfragen werden verteilt und sprechen mehrere Services an. Der API Gateway kann außerdem jedem Nutzer eine andere API zur Verfügung stellen und kann dadurch die entstehenden Bedürfnisse spezifischer handhaben. Es ist auch möglich die Autorisierung vom Client in den Gateway zu verschieben und somit den Client noch dünner zu Gestalten.

### Backends for frontends Bzw. Gateways je Nutzeroberflächen

Einen etwas anderen Ansatz nimmt die Variante welche „Backends for fronteneds“ 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. 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. Eine Autorisierung ist natürlich immer noch möglich.

<https://youtu.be/YO6Sg4yaqC0?t=185>

<https://blog.christianposta.com/microservices/api-gateways-are-going-through-an-identity-crisis/> API Gateway Pattern nochmals anschauen

Service discorvery compatible

Wo kann er unabhängig existieren und wo sich mit anderen (API Gateway & Service Mesh) ergänzen und co-existieren.

## API Microgateway(To Shallow? )

Der API Microgateway ist 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, wie verschiedene üblicherweise Traffic Management. 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 führt. 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.

## API Management

Die API Management oder der API Management Gateway ist eine Komponente, welche neben Bzw. über dem API Gateway steht. Sie 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. Alle APIs werden von diesem Management katalogisiert und verwaltet, wodurch diese vergeben, gefunden und effektive kontrolliert werden können.

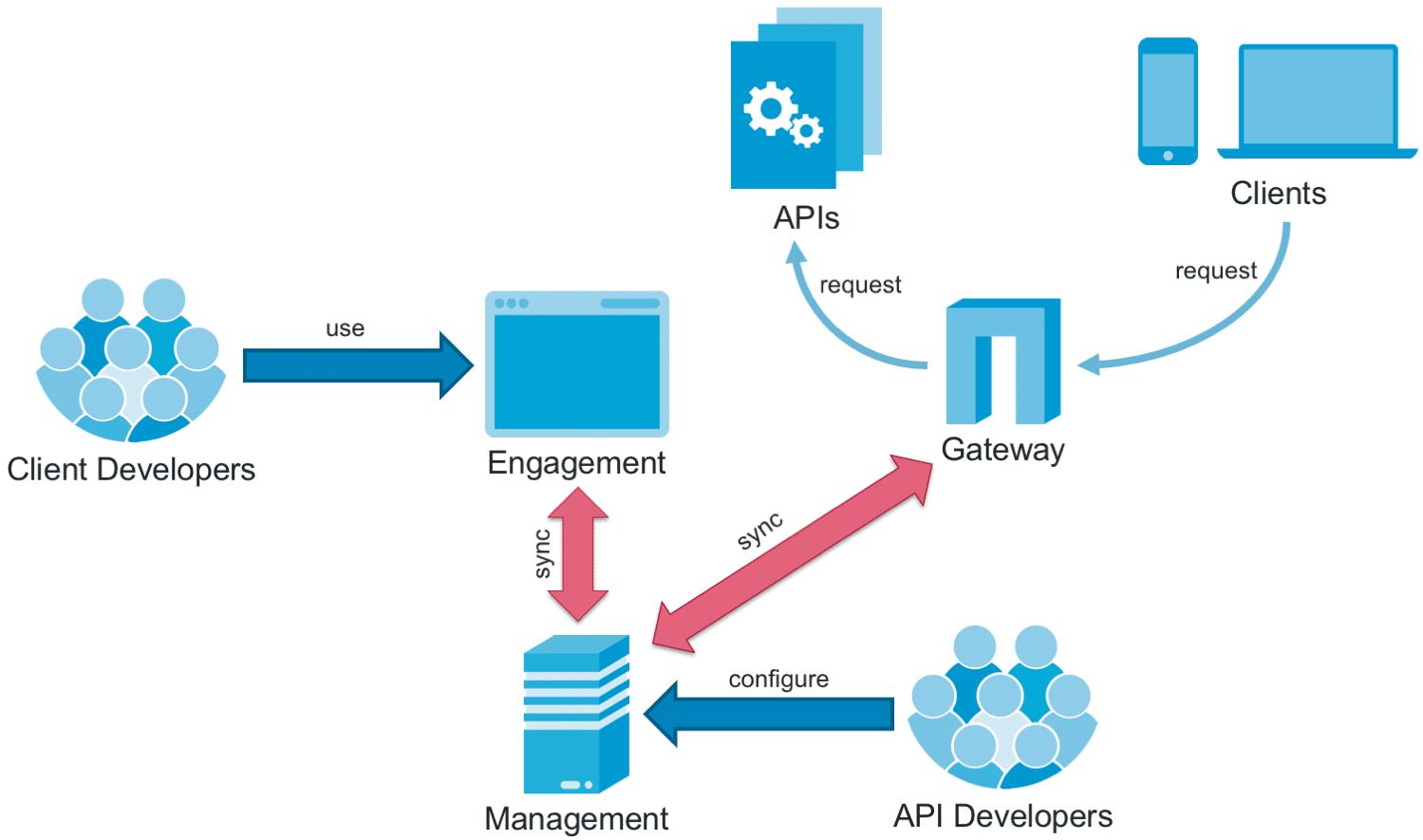


Figure 4: Komponenten einer API Management Lösung.jpg (10/24/2019)

Wie hier in Abbildung 3 zu sehen ist, ist die Managementkomponente, nicht diejenige über welchen jeglicher Nachrichtenaustausch mit den Nutzern geschieht, sondern stellt vielmehr den eigenen Entwicklern und dritt Entwicklern eine Möglichkeit dar mit den API zu interagieren und Änderungen an den Gateways durchzuführen.

Ein Punkt bei dem aufgepasst werden muss, ist keine Businesslogik in diese Schicht 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.

## Service-Mesh

<https://www.youtube.com/watch?v=cWfACBIp0a8>

### Warum Service-Meshes entstanden sind?

Mit einem Service-Mesh wird ein Netzwerk an Microservices beschrieben welche eine vollständige Anwendung und deren innere Kommunikation ausmacht. Durch die wachsende Größe und Komplexität dieser Service-Meshes, wird es immer schwieriger diese zu Überblicken und zu Managen. Es kommen alle möglichen Techniken zum Einsatz wie schon zuvor in Werkezuge oder Techniken für ein robustes verteiltes System aufgezeigt wurde. Es ist durchaus möglich dies alles auch ohne ein Service-Mesh zusammen zubauen oder sogar Notwendig wenn das Service-Mesh dies nicht anbieten. Hier müssen allerlei Aspekte verglichen werden und Natürlich kommt es auch immer auf die jeweilige Umsetzung des Anbieters an doch zu alledem später im Punkt 5 Umsetzung eines robusten verteilten Systems mit Hilfe eines Service-Meshes mehr.

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

Als Vorrausetzung für ein Service Mesh, ist eine Microservice Architektur, da sie genau für diese Umgebung gebaut sind und ansonsten Zuviel zusätzliche Arbeit verursachen würden. Ein Service Mesh, besteht normalerweise aus einem Zentralen Kontrollpunkt und vielen Sidecar Proxys, welche in den jeweiligen Container Instanzen mitlaufen aber kein Bestandteil des eigentlichen Prozesses sind. Die Proxys unterbrechen jeglichen Datenverkehr und kontrollieren bzw. Analysieren diesen. Der Kontrollpunkt erhält die Analysedaten und Verarbeitet diese. Selbst wenn ein Sidecar Proxy abstürzt kann die Kommunikation immer noch weitergehen, da die Kommunikation lediglich unterbrechen wird und nicht wie bei einem herkömmlichen Proxy als fester Zwischenpunkt agiert.

### Was Spricht gegen die Nutzung von Service Meshes

Service Meshes besitzen noch einige Nachteile, welche jeweils abgewogen werden müssen um Herauszufinden, ob es sich lohnt in diese schon zu Investieren. Demnach ist im Moment 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 Auswahl an unterschiedlichen Anbietern für Service Meshes macht dies umso schwieriger und macht eine Auswertung der Technologie ebenso unübersichtlicher. Man kann sich ein eigenes Bild z. B. über das Service Mesh Landscape (10/28/2019) machen.

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 wirfd.

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.

## Daten Konsistenz????

## Konfiguration Server

## Verteilte anfragen Verfolgung/Überwachung (Distributed tracing)

### Was ist verteilte anfragen Überwachung

Bei verteilter anfragen Überwachung 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 verteilte anfragen Verfolgung/Überwachung 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.



Figure 5: Why Distributed Tracing

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 verteilte Anfragenüberwachung 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“ (Dave McAllister 2019, 6:5-6:10). Es gibt einige „Standards“ 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" (Dave McAllister 2019, 15:15-15:25) da eine Codeanpassung für eine Anfragenüberwachung sehr umfangreich sein kann und niemand zweimal durchführen möchte. "Trust me instrumenting all of your code is more than enough" (Dave McAllister 2019, 15:37-15:42)

# 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?

#### Notizen

Helm install:

kubectl -n kube-system create serviceaccount tiller

kubectl create clusterrolebinding tiller-cluster-rule --clusterrole=cluster-admin --serviceaccount=kube-system:tiller

helm init --service-account tiller

helm init --upgrade --service-account tiller

Test:

kubectl -n kube-system rollout status deploy/tiller-deploy

helm version

Bin auf einen Helm Fehler mit der version 2.16.0 gestoßen welcher die Installation nicht möglich machte: Downgrade erforderlich

<https://github.com/helm/helm/issues/6894>

helm reset --force

helm init --service-account tiller --tiller-image gcr.io/kubernetes-helm/tiller:v2.14.3

dies downgraded nur die Server komponente die Client Komponente muss extra installiert und geändert werden.

Gelöst in 2.16.1

Install Istio Customizable Install with Helm:

kubectl apply -f install/kubernetes/helm/helm-service-account.yaml

helm install install/kubernetes/helm/istio-init --name istio-init --namespace istio-system

helm install install/kubernetes/helm/istio --name istio --namespace istio-system

Test:

kubectl get svc -n istio-system

kubectl get pods -n istio-system

kubectl get all -n istio-system

kubectl label namespace default istio-injection=enabled

kubectl get namespace -L istio-injection

Aktiviere Metrics und überwachung Nachträglich:

helm upgrade istio install/kubernetes/helm/istio --set grafana.enabled=true,kiali.enabled=true,tracing.enabled=true

Powershell Restrequest:

while ($true) {clear-Host;Invoke-RestMethod http://"$env:INGRESS\_HOST`:$env:INGRESS\_PORT"/productpage ;sleep 1}

# Zusammenfassung und Ausblick

# 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

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

Arnon Rotem-Gal-Oz: *Fallacies of distributed computing Explained.* URL http://www.rgoarchitects.com/Files/fallacies.pdf – Überprüfungsdatum 09.10.2019

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

Dave McAllister: *GOTO 2019 • Observability, Distributed Tracing & the Complex World • Dave McAllister - YouTube.* URL https://www.youtube.com/watch?v=2nTJSsBngao – Überprüfungsdatum 08.11.2019

Martin Fowler: *CircuitBreaker.* URL https://martinfowler.com/bliki/images/circuitBreaker/state.png. – Aktualisierungsdatum: 04.10.2019 – Überprüfungsdatum 14.10.2019

*Mit der richtigen API Management Architektur in die Cloud › ipt.* URL https://ipt.ch/mit-der-richtigen-api-management-architektur-in-die-cloud/. – Aktualisierungsdatum: 24.10.2019 – Überprüfungsdatum 24.10.2019

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

Stephan Augsten: *Was sind Container? : Definition „Container (Informatik)“.* URL https://www.dev-insider.de/was-sind-container-a-573872/. – Aktualisierungsdatum: 19.01.2017 – Überprüfungsdatum 10.04.2019

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

*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