Reaktive Microservices am Beispiel von Lagom

Tim Essig

Betreuer: Prof. Dr. Christian Zirpins

Zusammenfassung In dieser Seminararbeit werde die Eigenschaften und Vorteile reaktiver Microservices aufgezeigt. Es werden die drei Bestandteile Microservices, Reaktive Systeme und Reaktive Programmierung genauer betrachtet und erläutert, welche Vorteile diese Techniken gegenüber den bisherigen Herangehensweisen bieten. Um dies zu verdeutlichen wird anhand eines Beispielszenarios gezeigt, wie sich die beiden Ansätze in der Leistungsfähigkeit unterscheiden.

1 Einstieg

Die Anzahl der Internetnutzer steigt nahezu exponentiell, traditionelle Webanwendungen werden dadurch stetig wachsender Nutzung ausgesetzt die mit einem klassischen monolithischen Ansatz nicht mehr zu verarbeiten sind. Gleichzeitig nimmt auch die Zahl der mobilen Endgeräte zu, diese haben meist eine langsamere Netzwerkanbindung, so dass bei der Verwendung klassischer Threadpools die einzelnen Threads sehr lange belegt, bis alle Daten übertragen sind.

1.1 Herausforderungen Verteilter Systeme

Verteilte Systeme erlangten gemeinsam mit dem Erfolg des Internet an Bedeutung. Nicht nur zwischen *Client* und *Server* besteht eine Verbindung, auch zwischen den Servern untereinander entstehen Verbindungen. So liefert ein Webserver heutzutage nicht mehr nur statische HTML-Seiten aus, vielmehr ist dies dynamisch generierter Inhalt, welcher auf im Hintergrund laufende Datenbank-Server zugreift. Bei weiterer Skalierung einer Webseite wird meist die Datenbank als erstes zum Flaschenhals und so wird aus einem einzelnen Datenbank-Server schnell ein kleiner Cluster mit zwei oder mehr Datenbank-Server, welche sich die Last teilen.

CAP Theorem
Das CAP Theorem wurde ursprünglich im Jahr 2000 im Rahmen einer Keynote von Eric Brewer auf der ACM Symposium über die Prinzipien verteilten Computings vorgestellt[Brew00]. Laut Brewer kann ein verteiltes System höchstens zwei der drei folgenden Eigenschaften erfüllen.

Consistency Nach Abschluss einer Transaktion müssen auch alle Replikate den neuen Wert angenommen haben. Dies darf nicht mit dem Begriff der Konsistenz aus *ACID* verwechselt werden, welches lediglich die Konsistenz auf

Datenebene beschreibt(eine Transaktion wird entweder ganz oder gar nicht ausgeführt).

Availability Das System ist immer verfügbar(eng. available) und kann Anfragen stets innerhalb eines Timeouts beantworten.

Partition Tolerance Ein System aus mehreren Knoten kann auch bei teilweisem Ausfall dieser weiterarbeiten. Auch kann es weiterarbeiten, wenn das Netz durch Störungen in kleiner, unabhängige Teilnetze aufgespalten wird(partitioniert).

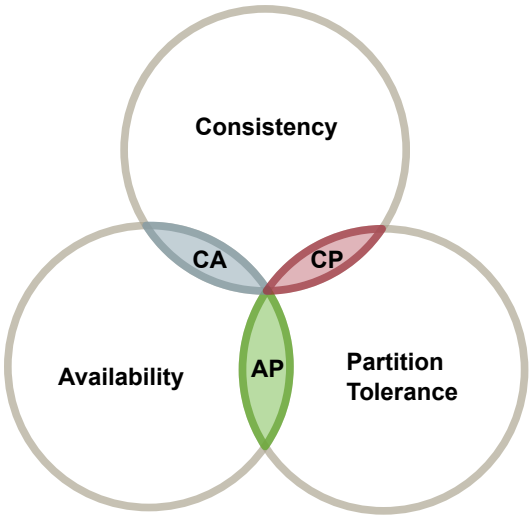


Abbildung 1. Verschiedene Eigenschaften die ein verteiltes System garantieren kann.[Erb12]

Verteilte Systeme und Datenbanken kann man meist in eine der folgenden drei Kategorien einteilen, abhängig davon, auf welche zwei Eigenschaften des CAP-Theorems sie sich spezialisieren.[1]

AP Das DNS ist ein weit bekanntes Beispiel für die Kategorie AP, da viel Wert auf hohe Verfügbarkeit und Resistenz gegen den Ausfall einzelner Knoten gelegt wird. Die Konsistenz ist relativ gering, bis sich eine Änderung über alle Knoten verteilt hat, können mehrere Tage vergehen. Auch die meisten Cloud-Technologien setzen auf AP, als Beispiel seien hier die meisten NoSQL

[1] `https://de.wikipedia.org/wiki/CAP-Theorem`

Datenbanken genannt, welche die BASE-Prinzipien verfolgen.

CA Relationale Datenbank Management Systeme (RDBMS) fallen meistens in die Kategorie CA. Sie streben hohe Verfügbarkeit und hohe Konsistenz der Daten an. Da diese Datenbank-Cluster meist in einem Rechenzentrum betrieben werden, können sich diese Systeme auf ein sehr stabiles Netz verlassen und müssen daher eher weniger resistent gegen Partitionierung sein.

CP Die wichtigste Eigenschaft von Bankensysteme ist die Konsistenz, ein Betrag, der am Geldautomat abgehoben oder eingezahlt wurde muss in jedem Fall auf dem Konto erscheinen. Auch gegenüber Störungen im Datenverkehr(Partitionierung) sollen Bankensysteme resistent sein. Dagegen ist die ständige Verfügbarkeit zweitrangig, wenn das System im Hintergrund nicht erreichbar ist, gibt der Geldautomat kein Geld aus.

Fallacies of Distributed Computing

In der Vergangenheit wurden Eigenschaften verteilter Systeme oft ignoriert. Bill Joy und Tom Lyon haben daher bei Sun Microsystems die Liste der *Fallacies of Distributed Computing*(dt. Falschannahmen verteilter Systeme) erstellt. Peter Deutsch hat diese 1994 um drei Punkte erweitert, James Gosling hat 1997 einen achten Punkt ergänzt[Rote06].

1. Das Netzwerk ist ausfallsicher
2. Die Latenzzeit ist gleich Null
3. Der Datendurchsatz ist unendlich
4. Das Netzwerk ist sicher
5. Die Netzwerktopologie wird sich nicht ändern
6. Es gibt immer nur einen Netzwerkadministrator
7. Die Kosten des Datentransports können mit Null angesetzt werden
8. Das Netzwerk ist homogen

Im weiteren Verlauf werden diverse Methoden und Entwurfsmuster vorgestellt, die versuchen mit genau diesen Problemen umzugehen.

1.2 Probleme bei Skalierung von Monolithen

In einem klassischen Monolithen sind alle Komponenten eng mit einander verzahnt und werden nur gemeinsam in Produktion gebracht(eng. deployed). Wenn diese Anwendung nun skaliert wird, müssen zwangsläufig alle Komponenten skaliert werden. Enthält eine Anwendung zum Beispiel eine Komponente um monatliche Reports zu generieren, so wird diese zwangsläufig mit skaliert, auch wenn dies nicht notwendig wäre um mehr Nutzer bedienen zu können. Die Resourcen werden also nicht optimal genutzt. Microservices hingegen ermöglichen es, diejenigen Services zu skalieren, die es benötigen.[Shar16]
Des weiteren neigen monotlithische System dazu, mit der Zeit eine immer stärkere Kopplung zu entwickeln, dieser Entwicklung in der Praxis entgegenzuwirken

bedarf einer großen Disziplin. Microservices hingegen forcieren auch hier wieder eine losere Kopplung der einzelnen Services.
Auch auf organisatorischer Ebene stoßen Monolithen bei der Skalierung an Grenzen, so müssen für große Anwendungen zum Teil hunderte Entwickler koordiniert werden, Test- und Release-Phasen müssen abgestimmt werden. Es muss sich auf eine Programmiersprache und meist auch ein Framework geeinigt werden, welches alle Belange der Anwendung abdecken können muss, jedoch gibt es selten ein Framework, welches alle Aufgaben gleich gut erfüllt. Auch hier bieten Microservices mehr Flexibilität indem ein Team von wenigen Entwicklern jeweils für eine kleine Menge von Microservices verantwortlich ist.

2 Microservices

Microservices ist ein Architekturstiel bei dem eine große monolithische Anwendung in einzelne unabhängige Services aufgeteilt wird, diese laufen in separaten Prozessen und können unabhängig voneinander in Produktion gebracht werden. Dieses vorgehen bringt zum einen Vorteile bei der Skalierung, da die einzelnen (Micro)Services in beliebiger Anzahl *deployed* werden können zum andern bringt dies auch organisatorisches Vorteile, da sich Teams nicht mehr auf einen Releasezyklus festlegen müssen, sondern ihre eigenen Services unabhängig voneinander entwickeln können.
Unterstützt wird die *Microservice* Architektur zudem auch durch das aufkommen des *Cloud-Computings*, welches sich bei Unternehmen steigender Beliebtheit erfreut[JaPa13]. Dies ermöglicht es die benötigte Rechenkapazität bei der Skalierung von *Microservices* schnell und kostengünstig bereit zu stellen und bei Überkapazität ungenutzte Rechenkapazität schnell wieder abzubauen.

2.1 Definition von Microservices

Die Idee der *Microservices* ist nicht neu. Einen ganz ähnlichen Ansatz verfolgt schon die UNIX Philosophie. Sie basiert auf drei Ideen:[Wolf17]

- Ein Programm soll nur eine Aufgabe erfüllen
- Die Programme sollen zusammenarbeiten können.
- Die Programme sollen eine universelle Schnittstelle nutzen. In UNIX sind dies Textströme.

Dadurch entstehen einzelne Komponenten,die miteinander Kombiniert werden können. Diese Lose Kopplung und Isolation gegenüber anderen Komponenten zeichnen *Microservices* aus.

2.1.1 Lose Kopplung und hohe Kohäsion

Hauptziel der Modularisierung durch *Microservices* ist es, dass die einzelnen Services möglichst unabhängig voneinander sind. Durch lose Kopplung wird erreicht, dass ein Service verändert und deployed werden kann, ohne dass hierbei

andere Teile des Systems geändert werden müssen. Um dies zu erreichen dürfen sich *Microservices* möglichst wenige Dinge teilen. Zum einen dürfen sich *Microservices* keine Datenbank teilen, da eine Änderung am Datenbankschema auch andere *Microservices* beeinflussen würden. Zum anderen darf auch kein Quelltext geteilt werden. Hierbei wird bewusst gegen das DRY-Prinzip(don't repeat yourself, deutsch: wiederhole dich nicht) verstoßen[Newm15].

Weiterhin sollte aller ähnlicher Code in einem Service gebündelt sein, Code welcher andere Funktionen erfüllt, sollte in einen eigenen Service ausgelagert sein. Somit wird ermöglicht, dass eine Änderung, zum Beispiel an der Benutzerverwaltung, nur an einer Stelle umgesetzt werden muss. Diese Änderung lässt sich dann ebenfalls unabhängig von anderen Services deployen. Somit können Änderungen schneller umgesetzt werden, als wenn diese an mehreren Stellen umgesetzt werden müssten. Des weiteren muss so nur ein Service neu deployed werden und nicht vielleicht sogar mehrere gleichzeitig, was sehr riskant ist. Man spricht hierbei von hoher Kohäsion.
Daher müssen Schnittstellen gefunden werden, die sicher stellen, dass sich der, für zusammengehöriges Verhalten zuständiger Code, nur an einer Stelle befindet und dafür sorgen, dass dieser über Schnittstellen möglichst stark von anderem Code entkoppelt ist.

2.1.2 Technologiefreiheit

Ein großer Vorteil der Microservice-Architektur ist die Technologiefreiheit. Da die einzelnen Services über ein definierte Schnittstelle(oft RESTful Webservices/HTTP) miteinander kommunizieren, sind die einzelnen Services nicht an eine bestimmte Programmiersprache oder gar Framework gebunden. Services können in jeder Technologie implementiert werden, die den genutzten Kommunikationsweg unterstützt. Im Fall von RESTful Webservices dürfte dies auf nahezu jede Programmiersprache zutreffen. Somit kann auch eine technologische Grundsatzentscheidung der Geschäftsleitung einfacher umgesetzt werden, falls zum Beispiel von *.NET* auf *Java EE* oder *Ruby on Rails* umgestellt werden soll, so können neue Services realisiert werden, ohne von der Technologie der bereits existierenden Services abhängig zu sein.

2.2 Microservice Architektur und Patterns

Nachfolgend sollen eine Auswahl der gängigsten Entwurfsmuster(eng. Patterns) aus dem Bereich der Microservices vorgestellt werden.

2.2.1 Bounded Context

Bounded Context ist ein Entwurfsmuster oder Konzept aus dem Bereich des Domian-Driven-Designs(kurz DDD). Im Gegensatz zur klassischen Entwicklung die mit einem ER-Model für die Datenbank beginnt, welches für die ganze Anwendung gleich ist, geht DDD den Weg, dass Verschiedene Fachbegriffe in verschiedenen Domänen unterschiedliche Bedeutungen haben können [Vern17]. Ein

Beispiel hierfür wäre ein Artikel in einem produzierenden Betrieb. Im Kontext der Produktion hat dieser Artikel Informationen wie z.B. Stücklisten, diese sind für den Vertrieb irrelevant, im Kontext des Vertriebs sind für den Artikel Informationen relevant wie zum Beispiel der Preis und die Verfügbarkeit. Meist wird der Stammsatz eines Artikels daher mit vielen Feldern angereichert, die nur in einem von vielen Kontexten benötigt werden. In vielen ERP-Systemen wird diesem Problem mit Views begegnet, dies löst jedoch nicht die starke Abhängigkeit der unterschiedlichen Kontexte über das geteilte Datenbankschema.
Im DDD werden diese beiden Kontexte nun getrennt modelliert und der Artikel als Schnittstelle definiert. In beiden Kontexten existiert nun eine Entität Artikel, welche allerdings unterschiedliche Felder besitzen. Teilen tun sich diese einen eindeutigen Identifier, mit dem sie einander zuordenbar sind.

2.2.2 Circuit Breaker

Es liegt in der Natur Verteilter Systeme, dass andere System aufgerufen und auf eine Antwort dieser gewartet wird. Hierbei kann es zu verschiedenen Fehlern kommen, zum einen können Probleme mit der Verbindung auftreten(Netzwerk unterbrochen, Timeout,...) oder das entfernte Systeme ist einfach abgestürzt oder überlastet und reagiert daher nicht innerhalb der nötigen Zeit.
Um zu vermeiden, dass unnötige Anfragen an ein nicht (rechtzeitig) reagierendes System gesendet werden, wird das *Circuit Breaker* Entwurfsmuster eingesetzt(dt. Schutzschalter). Dieser überwacht die Aufrufe an ein entferntes System, antwortet dieses nicht oder nicht innerhalb eines Timeouts, so öffnet sich der Lastschalter und unterbindet dadurch, dass die Anfragen an das entfernte System gerichtet werden. Dies hat zwei Vorteile, zum einen weiß das aufrufende System sofort, dass es keine Antwort erhält und kann einen anderes System anfragen. Zum andern wird das entfernte System, welches unter Umständen zur Zeit einfach nur überlastet ist und daher langsam reagiert, vor weiteren Anfragen geschützt und hat so die Möglichkeit, seine anstehenden Abfragen abzuarbeiten und anschließend wieder schneller antworten zu können.

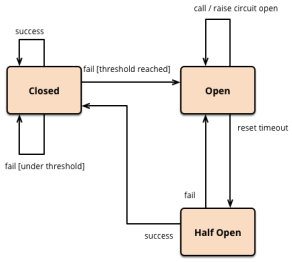
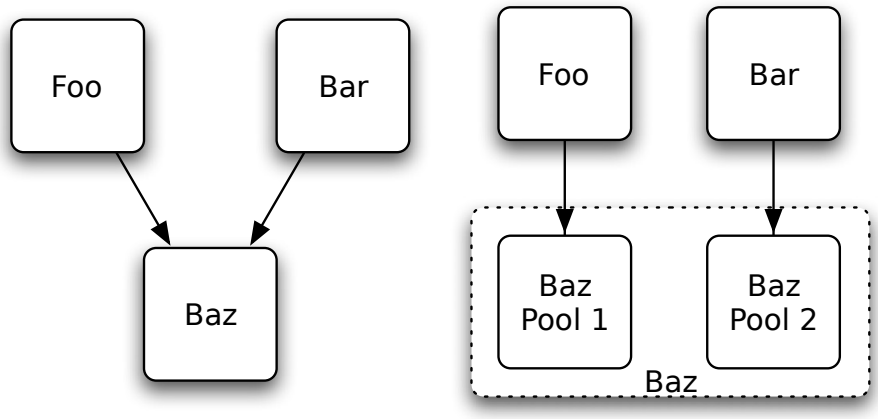


Abbildung 2. Zustandsdiagramm eines Circuit Breaker [2]

Der *Circuit Breaker* bleibt allerdings nicht für immer geöffnet, nach einem definierten Timeout, wird erneut eine Anfrage an das betreffende System weitergegeben, der Cirucit Breaker ist nun im Zustand "Half-Open". Ist diese Anfrage erfolgreich, so wird der Circuit Breaker wieder geschlossen und Anfragen werden wieder ganz normal an das entfernte System weitergegeben. Schlägt diese Anfrage allerdings fehl, so bleibt der Circuit Breaker geöffnet und nach einem definierten Timeout wird erneut getestet, ob das System nun wieder antwortet. Das entsprechende Zustandsdiagramm ist in Abbildung 2 zu sehen.

2.2.3 Bulkhead

Das *Bulkhead-Pattern*(dt. Trennwand) wurde ursprünglich im Schiffs- und Flugzeugbau eingesetzt um einzelne Bereiche des Schiff oder Flugzeugs abschotten zu können, falls es eine Undichtigkeit in der Bordwand gibt. Somit wird vermieden, dass das ganze Schiff voll Wasser läuft und sinkt. Ein weit verbreitetes Beispiel, wo dies nicht funktioniert hat, ist die Titanic.
In der Software-Entwicklung kann dieses Entwurfsmuster ähnlich angewendet werden um zu vermeiden, dass der Ausfall eines einzelnen Systems andere Systeme negativ beeinflusst. Einzelnen (Gruppen) von Klienten wird ein bestimmte Menge Resourcen zur Verfügung gestellt(Abbildung 3).



(a) Verborgene Abhängigkeit (b) Partitioniertes System

Abbildung 3. Anwendung des *Bulkhead* Entwurfsmusters[Nyga07]

Nachfolgend sind zwei Beispiel aus dem Buch *Release it!* aufgezählt.[Nyga07]

Redundanz Durch Redundanz des Systems wird verhindert, dass ein Fehler in einem Teil das ganze System beeinflusst. Ein bekanntest Beispiel ist die

physikalische Redundanz. Wenn von vier redundanten Servern hinter einem *Load-Balancer* ein Server ausfällt, hat dies keinen Einfluss auf die Funktionalität der anderen Server.
Laufen auf auf einem Server zwei unabhängige Instanzen einer Anwendung, so hat der Absturz einer Instanz keinen Einfluss auf die andere.

Partitionierung Bei großen verteilten Systemen ist es Sinnvoll, kritische von unkritischen Systemen zu trennen. Ein hoch frequentierter Service zum abfragen des Flugstatus ist weniger kritisch als der weniger hoch frequentierte Service zum einchecken der Passagiere am Flughafen. Daher sollte ein Ausfall des Flugstatus-Service die Funktionsfähigkeit des Check-In Service nicht beeinträchtigen. Daher sollten diese auf getrennten Servern betrieben werden, so dass zum Beispiel, bei stürmischem Wetter, wenn der Flugstatus-Service durch viele Anfragen überlastet ist, die Server für den Check-In weiterhin funktionieren. Verborgene Abhängigkeiten werden so vermieden.

2.2.4 Service Registry

Damit die vorher genannten Entwurfsmuster sinnvoll umzusetzen sind muss es von jedem Service mehrere Instanzen geben. Es stellt sich nun die Frage, wie der Anfragende Service einen der vielen Services finden kann. Würde er einen DNS-Namen auflösen oder gar auf eine feste IP-Adresse zugreifen, wäre die Implementierung sehr starr. Hätte jeder Service eine Liste mit möglichen IP-Adressen könnte sich dieser zwar zwischen verschiedenen Instanzen entscheiden und damit einen *Circuit-Breaker* oder *Bulkhead* realisieren, ändert sich aber eine dieser IP-Adressen oder würden dynamisch neue Instanzen gestartet werden, so müsse der Service mit den geänderten IP-Adressen neu deployed werden.
Als Lösung für dieses Problem werden *Service Registries* eingesetzt. Diese verwaltet die IP-Adressen sowie Ports der einzelnen Instanzen. Hierzu muss jeder Service einen *Registry Client* nutzen, um sich bei der *Service Registry* anzumelden. Es ist sinnvoll, dass die *Service Registry* ebenfalls einen *Health-Check* durchführt um eventuell nicht mehr existierende Instanzen automatisch zu entfernen.

Vorgegangen wird nun wie in Abbildung 4 gezeigt. Möchte Service A nun Service B aufrufen, so stellt dieser zunächst eine Anfrage an die *Service Registry* um von dieser eine IP-Adresse und Port einer Instanz zu bekommen und stellt anschließend ihre Anfrage an die erhaltene Instanz.
Bekannte Implementierungen einer *Service Registry* sind zum Beispiel Eureka von Netflix[4] oder ZooKeeper von Apache[5].

2.2.5 Deployment

Gemeinsam mit der Microservice-Architektur gewann auch Docker[6] an Beliebtheit. Mit Hilfe von Docker lassen sich Anwendungen in *Container* verpacken.

[4] `https://github.com/Netflix/eureka`
[5] `https://zookeeper.apache.org/`
[6] `https://www.docker.com/`

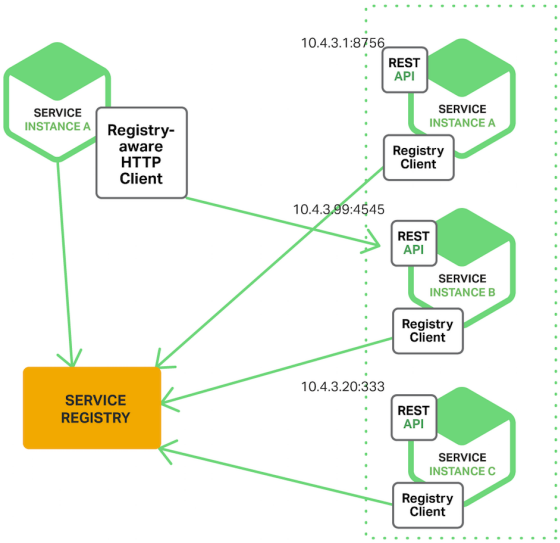


Abbildung 4. Zustandsdiagramm eines Circuit Breaker [3]

Ein Prozess der in einem *Container* läuft, ist von den anderen Prozessen des Host-Systems unabhängig. Ein *Container* ist keine Virtuelle-Maschine, vielmehr lässt er sich mit dem UNIX-Befehl `chroot` vergleichen. Daher stellt Docker eine Virtualisierung auf Ebene des Betriebssystems dar, die virtuellen Umgebungen laufen direkt auf dem Kernel des Host-Systems und werden mithilfe von verschiedenen Mechanismen vom Rest des Systems abgeschottet. Dieser Ansatz ist deutlich leichtgewichtiger als andere Virtualisierungstechniken wie zum Beispiel XEN oder VMware. Da Docker kein komplettes Betriebssystem virtualisieren muss, reduziert sich der Bedarf an Arbeitsspeicher enorm, auch das Starten eines *Containers* geschieht in einem Bruchteil der Zeit. Hierdurch können auf der gleichen Hardware wesentlich mehr Instanzen einer Anwendung gestartet werden, als dies mit klassischen Virtualisierungslösungen möglich ist. Docker war zunächst nur für Linux verfügbar, mittlerweile bietet jedoch auch Microsoft immer bessere Unterstützung für *Container*-Technologien.
Docker verwendet für *Container* ein Dateisystem, welches mit Ebenen(englisch Layer) arbeitet. Jede Änderung beim erzeugen eines Images wird in einer neuen Ebene abgelegt. Vereinfacht betrachtet kann ein Image damit zum Beispiel aus den Ebenen Betriebssystem, Applicationserver, Applicationserver-Config bestehen. Soll nun ein Image für eine Anwendung erstellt werden, so wird die WAR-Datei dieser Anwendung in das Deployment-Verzeichnis des Applicationservers kopiert, somit entsteht ein neuer Layer. Die unteren Layer für Betriebssystem und Applicationserver können von jedem anderen Image wiederverwendet werden. Somit müssen diese, meist großen, Images nur einmal auf der Maschine liegen und können für jede Anwendung wieder verwendet werden.[MoDe16]

Obwohl es möglich wäre, mehrere Prozesse in einem *Container* auszuführen, empfiehlt die Docker-Community, genau einen Prozess pro *Container* auszuführen. So bleiben zum Beispiel die Webanwendung und die Datenbank in zwei separaten *Container*, so dass je nach Bedarf zusätzliche *Container* vom einen oder anderen Typ gestartet werden können. Docker Images werden aus *Dockerfiles* erzeugt, diese definieren, wie die Images konfiguriert werden. Damit sind im *Container* auch alle Abhängigkeiten der Anwendung enthalten. Die Anwendung kann über Netzwerk(freigegeben Ports) und geteilte Ordner mit der Außenwelt kommunizieren. Damit stellt ein *Container* ein Self-Contained-System dar, welches eigenständig lauffähig ist. Somit kann das Image auf dem lokalen Entwickler-PC genau so laufen wie auf dem Produktiv-Server. Freigegebene Ports können nach außen auf beliebige Ports gemappt werden, somit gibt es hierbei keine Überschneidungen, eine *Service Registry* muss in diesem Fall verwalten, welcher Service unter welchem Port zu erreichen ist.
Da die Images *Self-Contained* sind, können nun für das horizontale Scale-Out beliebig viele Instanzen parallel gestartet werden und so die Kapazität des Systems erhöht werden. Verringert sich die Auslastung des Systems, können Instanzen gestoppt werden um nicht unnötig Resourcen zu belegen, gerade bei Cloud-Angeboten werden hiermit direkt Kosten eingespart.

2.3 Vor- und Nachteile

Eine Microservice-Architektur bekommt man selbstverständlich nicht umsonst. Einzelne Services werden zwar kleiner und übersichtlicher, es wird aber mehr Komplexität in die Infrastruktur ausgelagert. Auch darf der Überblick über die vielen kleinen Services nicht verloren gehen, was auch einen organisatorischen Aufwand mit sich bringt.
Durch das Aufteilen in einzelne Services ist es möglich, die Teile der Anwendung zu skalieren, die dies benötigen. In Kombination mit Container kann so schnell und effizient horizontal skaliert werden. Diese, am besten automatisierte Skalierung, bedarf jedoch der Orchestrierung des Systems mithilfe spezieller Tools wie zum Beispiel Kubernetes[7].
Im Zusammenhang mit Microservices hört man auch oft Conway's Law[Conw68]. Dieses besagt, dass sich die Organisationsstruktur eines Unternehmens und die Software-Architektur gegenseitig beeinflussen. Werden Microservices entwickelt, so macht es Sinn, die Teams entlang von Microservice-Grenzen zu schneiden. Wurde zuvor ein Monolith entwickelt, und die Teams waren nach Aufgabenbereich aufgeteilt(z.B. Frontend, Backend, Datenbank), so müssen nun Teams erstellt werden, welche mindestens einen Entwickler aus jedem Bereich haben, um einen eigenständigen Microservice zu entwickeln.
Durch die kleineren Services die unabhängig voneinander getestet und deployed werden können, verringert ich auch die *Time to Market*. Ein Fehler in einem Teilmodul blockiert nicht länger den Release der ganzen Anwendung.

[7] `https://kubernetes.io/`

Eine Microservice Architektur bringt also nicht nur Vorteile, sondern auch einige Nachteile. Daher muss sich die Frage gestellt werden, ob die Vorteile, die mir diese liefert, wirklich die Nachteile aufwiegen. Meist lohnt sich dies erst ab einer gewissen Größe der Anwendung.

3 Reaktive Systeme

Im vorherigen Kapitel haben wir *Microservices* als eine Lösung für die Probleme eines Monolithen kennen gelernt. Durch die einzelnen Instanzen eines Service kann das System besser skaliert werden.
Bei steigender Auslastung des Systems, fällt allerdings ein weiterer Flaschenhals auf. Durch die synchrone REST basierte Kommunikation warten die einzelnen Services in Summe doch sehr lange auf eine Antwort. Daten müssen übertragen und empfangen werden, die Datenbank muss auf das Ende einer laufenden Transaktion warten, etc. dies sind nur wenige Beispiel in denen der Service wartet und diese Zeit aber effektiver nutzen könnte.

3.1 Reaktive Manifesto

Die Idee der Reaktiven Systeme wird im Reactive Manifesto 2.0 beschrieben. Verfasst wurde dies unter anderem von Jonas Bonér welcher ebenfalls das Aktoren-Framework *Akka* erfand. Die Autoren vertreten die Meinung, dass die Software-Entwicklung, wie sie in den letzten 15 Jahren betrieben wurde, den heutigen Anforderungen nicht mehr gerecht werden. Die Systeme sind in jeder Hinsicht gewachsen, die Zahl der Server und Prozessorkerne nahm zu, Speicherplatz und Arbeitsspeicher werden zunehmend billiger. Mit aufkommen des Cloud Computings können einfach hunderte Server zu einem System hinzugefügt und entfernt werden. Zudem hat sich die Erwartung der Nutzer verändert, war es vor Zeiten von DSL und mobilem Internet noch üblich mehrere Sekunden auf die Antwort einer Webseite zu warten, so werden heutzutage Reaktionszeit im Millisekunden Bereich gemessen und erwartet. Um Systeme zu entwickeln, die diesen Anforderungen gerecht werden, haben Jonas Bonér, Dave Farley, Roland Kuhn und Martin Thompson folgende vier Eigenschaften als grundlegend definiert[8].

Reaktiv(eng. responsive)
: Das System antwortet unter allen Umständen zeitgerecht, solange dies überhaupt möglich ist. Antwortbereitschaft ist die Grundlage für Funktion und Benutzbarkeit eines Systems, aber noch wichtiger ist, dass Fehler in verteilten Systemen nur durch die Abwesenheit einer Antwort sicher festgestellt werden können. Ohne vereinbarte Antwortzeitgrenzen ist die Erkennung und Behandlung von Fehlern nicht möglich. Eine weitere Facette ist, dass konsistente Antwortzeiten als Zeichen von Qualität Vertrauen stiften und so weitere Interaktion fördern.

[8] Abgerufen am 28.5.17: `http://www.reactivemanifesto.org/de`

Widerstandsfähig(eng. resilient)
Das System bleibt selbst bei Ausfällen von Hard- oder Software antwortbereit. Dies bezieht sich nicht nur auf hochkritische Anwendungen: jedes System, welches nicht widerstandsfähig ist, verliert durch einen Ausfall seine Antwortbereitschaft und damit seine Funktion. Widerstandsfähigkeit ist nur erreichbar durch Replizieren der Funktionalität, Eindämmung von Fehlern, Isolation von Komponenten sowie Delegieren von Verantwortung. So bleibt der Ausfall eines Teilsystems auf dieses begrenzt, andere Teilsysteme sind geschützt und in ihrer Funktion nicht behindert. Die Wiederherstellung des Normalzustandes wird einer übergeordneten Komponente übertragen, die durch die gesteuerte Replizierung der ihr untergeordneten Komponenten die geforderte Verfügbarkeit sicherstellt. All dies bedeutet, dass Nutzer eines auf diese Weise widerstandsfähigen Systems von der Last befreit sind, sich mit dessen Ausfällen auseinandersetzen zu müssen.

Elastisch(eng. elastic)
Das System bleibt auch unter sich ändernden Lastbedingungen antwortbereit. Auch hier bildet die Verteilung und Replizierung von Funktionalität die Grundlage, auf der das System auf Veränderungen reagiert. Bei Verminderung oder Erhöhung der Last werden automatisch die Replizierungsfaktoren und damit die genutzten Ressourcen angepasst. Um dies zu ermöglichen, darf das System keine Engpässe aufweisen, die den Gesamtdurchsatz vor Erreichen der geplanten Maximalauslegung einschränken. Ideal ist eine Architektur, die keine fixen Engpässe aufweist. In diesem Fall kann das bearbeitete Aufgabengebiet in unabhängige Teile zerlegt und auf beliebig viele Ressourcen verteilt werden. Reaktive Systeme unterstützen die Erfassung ihrer Auslastung zur Laufzeit, um automatisch regelnd eingreifen zu können. Dank ihrer Elastizität können sie auf Speziallösungen verzichten und mit handelsüblichen Komponenten implementiert werden.

Nachrichtenorientiert(eng. Message-Driven)
Das System verwendet asynchrone Nachrichtenübermittlung zwischen seinen Komponenten zur Sicherstellung von deren Entkopplung und Isolation sowie zwecks Übermittlung von Fehlern an übergeordnete Komponenten. Die explizite Verwendung von Nachrichtenübermittlung führt zu einer ortsunabhängigen Formulierung des Programms und erlaubt die transparente Skalierung von Komponenten. Die Überwachung von Nachrichtenpuffern ermöglicht kontinuierlichen Einblick in das Laufzeitverhalten des Systems — sowohl zur Diagnose als auch zur automatischen Ressourcensteuerung — sowie Priorisierung und Kontrolle der Nachrichtenflüsse. Ortsunabhängigkeit bedeutet, dass Code und Semantik des Programms nicht davon abhängen, ob dessen Teile auf demselben Computer oder verteilt über ein Netzwerk ausgeführt werden. Nicht-blockierende nachrichtenorientierte Systeme erlauben eine effiziente Verwendung von Ressourcen, da Komponenten beim Ausbleiben von Nachrichten vollständig inaktiv bleiben können.

Mit Hilfe dieser vier Eigenschaften ist es möglich, Systeme zu bauen, welche eine hohe Flexibilität und Ausfallsicherheit aufweisen. Durch ihren Aufbau können

diese hervorragend skalieren. Durch die ständige Antwortbereitschaft sind diese System besonders Benutzerfreundlich.
Diese Eigenschaften müssen durch die ganze Architektur hinweg berücksichtigt werden,

4 Reaktive Programmierung

4.1 Motivation (Probleme von Threads)

Die asynchrone Kommunikation die Reaktive Systeme nutzen verbessert bereits die Leistungsfähigkeit der *Microservices*, allerdings kommen immer noch Threads zum Einsatz, diese können zwar nun warten, bis ihre Antwort angekommen ist, jedoch führt dies zu vielen Kontextwechseln, welche bei hoher Anzahl teuer sind. Reaktive Programmierung setzt nun darauf, Aufgaben in kleinere Teilaufgaben zu zerlegen, die asynchron berechnet werden können, die Ergebnisse dieser Berechnungen können anschließend zusammen gefügt werden, es wird aus den einzelnen Blöcken ein Workflow erstellt. Reaktive Programmierung erlaubt es nun, diese Teilaufgaben in einer nicht blockierenden Art auszuführen, damit müssen keine Threads auf Ergebnisse warten, sondern können so lange andere Aufgaben abarbeiten. Abbildung 5 stellt die unterschiedliche Threadnutzung des blockierenden und des reaktiven Ansatzes dar.

Reaktive Programmierung ist Event-getrieben, wohingegen Reaktive Systeme Nachrichten-getrieben sind, darauf wird später noch eingegangen. Die API's von Reaktiven Bibliotheken sind meist entweder

Callback bassierend Anonyme Funktionen mit Seiteneffekten werden an Ereignisquellen geknüpft und aufgerufen, wenn Ereignisse in der Datenflusskette auftreten.

Deklarativ Durch das kombinieren von Funktionen wie zum Beispiel *map, filter, reduce*

Die meisten Bibliotheken bieten eine Mischung aus beidem. Reaktive Programmierung hängt auch mit Dataflow Programming zusammen, da der Schwerpunkt auf dem Datenfluss, nicht auf dem Kontrollfluss, liegt.
Bekannte Bibliotheken, welche Reaktive Programmierung ermöglichen sind: Ratpack, Reactor, RxJava, Vert.x und Akka. Auf letzteres wird im weiteren Verlauf eingegangen, da Lagom Akka verwendet.
Reaktive Programmierung konzentriert sich auf kurzlebige Datenflussketten und sind meist Ereignisgesteuert wohingegen Reactive Systeme sich auf Resilienz und Elastizität konzentrieren und Nachrichtengetrieben sind.
Der Hauptunterschied dieser beiden Ansätze ist der, dass Nachrichten einen bestimmten Empfänger haben, Ereignisse hingegen werden einfach veröffentlicht und eine beliebige Anzahl Empfänger kann auf diese reagieren. Nachrichten sind in Idealfall auch asynchron, was Sender und Empfänger weiter voneinander entkoppelt.
Nachrichtengetriebene Systeme sind auf adressierbare Empfänger ausgerichtet,

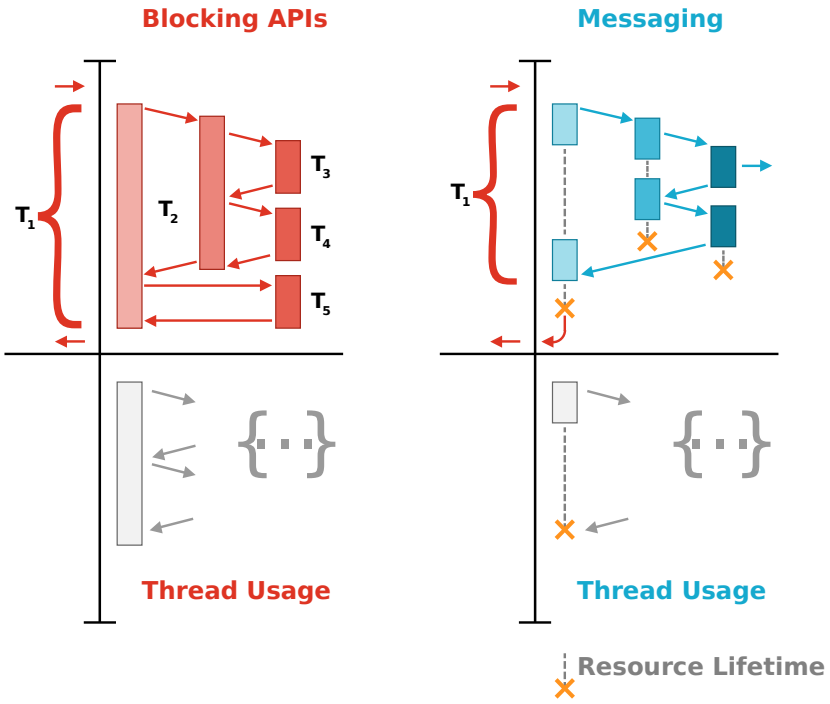


Abbildung 5. Synchrone Kommunikation(links) ist Resource hungrig. Ein reaktiver Ansatz(rechts) belegt die Resourcen weniger unnötig und nutzt die Hardware somit besser aus.

ereignisgesteuerte Systeme auf adressierbare Sender. Nachrichten werden über das Netzwerk verteilt, wohingegen Ereignisse lokal erzeugt und behandelt werden. In einem reaktiven System, welches auf reaktive Programmierung setzt, sind beide Arten vorhanden. Ereignisse können mittels Nachrichten über das Netzwerk gesendet werden. Jedoch muss bei der Verwendung von Nachrichten an die Probleme verteilter Systeme gedacht werden, so ist die Reihenfolge der Nachrichten nicht sicher gestellt oder das diese nur einmal empfangen wird, auch können Nachrichten verloren gehen. All diese potenziellen Fehler müssen bedacht und entsprechend behandelt werden.

4.2 Entwicklung von synchron zu asynchron in Java

Mit dem synchronen Ansatz sind die meisten Entwickler vertraut. Wie schon gezeigt wurde, entstehen hierbei allerdings lange Wartezeiten in denen ein Thread

blockiert. Speziell bei Kommunikation über Netzwerk, ist eine gewisse Latenz vorhanden(vgl. *Fallacies of Distributed Computing* Kapitel 1.1). Daher muss die Zeit, in der auf eine Antwort gewartet wird, sinnvoll genutzt werden. In den meisten Fällen gibt man den Thread an das genutzt Framework zurück, welchen diesen wieder anderweitig nutzt.
Zur Veranschaulichung wählen wir das Beispiel eines Pay-TV Anbieters[Bate16]. Es soll geprüft werden, ob ein Benutzer die Berechtigung hat, einen bestimmten Kanal zu schauen.
Zunächst muss der *UserService* den User für die übergebene User-ID liefern, anschließend kann der *PermissionService* prüfen, ob der Benutzer die Berechtigung besitzt. Zudem muss geprüft werden, ob der angefragte Kanal existiert. Jede dieser REST-Aufrufe benötigt in diesem Beispiel 500ms.

```
1  @Path("tv")
2  public class TvService {
3     @Inject
4     UserService us;
5
6     @Inject
7     PermissionService ps;
8
9     @Inject
10    ChannelService cs;
11
12    @GET
13    @Path("watch-channel/{username}/{permission}/{channel}")
14    public boolean watchChannel(@PathParam("username")
15                        String username,
16                        @PathParam("permission")
17                        String permission,
18                        @PathParam("channel")
19                        String channel) {
20
21       User user = us.getUser(username); //500ms
22       boolean hasPermission =
             ps.getPermissions(user).hasPermission(permission); //500ms
23       Channel c = cs.getChannel(channel); //500ms
24
25       return hasPermission && c != null;
26    }
27 }
```

Gut zu erkennen ist, dass bei synchroner Programmierung in Summe 1500ms auf die Antworten gewartet wird. Wenn man die Abhängigkeiten genauer betrachtet sieht man, dass der Aufruf des *PermissionService* von dem *UserService* abhängt, der Aufruf des *ChannelService* kann jedoch parallel erfolgen.
Es gibt nun die Möglichkeit, denn Aufruf in einen eigenen Thread auszulagern, mit dem Interface *Runnable* ist dies einfach möglich, jedoch hat *Runnable* kei-

nen Rückgabewert. Daher behelfen wir uns dem etwas neueren Interface *Callable<V>* welches einen Rückgabewert anbietet. Das *Callable* wird nun an einen *ExecuterService* übergeben, welcher dies in einem anderen Thread parallel ausführt. Da Callable ein *@FunctionalInterface*[9] ist, können wir den Code durch die Verwendung eines Lambdas vereinfachen.

```
ExecuterService es = Executors.newCachedThreadPool();
Future<Channel> f = es.submit(()-> cs.getChannel(channel));//Callable
```

Der Aufruf des *ChannelService* erfolgt nun parallel und blockiert nicht den weiteren Verlauf der Funktion. Das Resultat des *Callable* erhalten wir mittels `Channel c = f.get();`. Diese Funktion blockiert so lange, bis das *Callable* zurückspringt. Ist dies bereits geschehen, wird der Rückgabewert gespeichert und bei Aufruf von get() zurück gegeben. Die Funktion kann nun parallelisiert werde:

```
1  @GET
2  @Path("watch-channel/{username}/{permission}/{channel}")
3  public boolean watchChannel(/*...*/)) {
4     ExecuterService es = Executors.newCachedThreadPool();
5     Future<Channel> fChannel = es.submit(() -> cs.getChannel(channel));
6
7     User user = us.getUser(username); //500ms
8     boolean hasPermission =
          ps.getPermissions(user).hasPermission(permission); //500ms
9     Channel c = fChannel.get(); //Ergebnis einsammeln
10
11    return hasPermission && c != null;
12 }
```

Nun werden alle parallelisierbaren Aufrufe auch parallel abgearbeitet, der Thread ist allerdings weiterhin für 1000ms belegt, obwohl nichts getan wird, außer auf Ergebnisse zu warten. Dies würde sich auch nicht ändern, wenn jede Abfrage in einen eigenen Thread ausgelagert wird, da die Ergebnisse immer noch im Eltern-Thread zusammengefügt und zurück gegeben werden müssten.

In Java SE 8 kam das *CompletableFuture<T>* hinzu. Hiermit lässt sich dieses Problem angehen, es kann nun, zum Beispiel im Vergleich mit dem reinen Arbeiten mit *Future<T>*, relativ einfach eine verkettete Abfolge von Tasks definiert werden, die asynchron ausgeführt werden sollen.
Mit `CompletableFuture::thenCombine` können zwei *CompletableFuture* zusammengeführt werden.
`CompletableFuture::thenCompose` übergibt das Ergebnis an ein weiteres *CompletableFuture*.
Zum Schluss wird das Resultat mit `CompletableFuture::thenAccept` angenommen und wiederum asynchron verarbeitet. an dieser stelle wird das Ergebnis

[9] https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html

mittels `AsyncResponse::resume` zurück gegeben.

Seit JAX-RS 2.0 gibt es einen *AsyncResponse*, mit diesem kann der Rückgabewert asynchron zurück gegeben werden. Die Methode bekommt den Rückgabetyp void. Die Berechnung des Rückgabewerts kann in einem separaten Thread erfolgen und wird mit `AsyncResponse::resume( returnValue );` zurück gegeben. Die Funktion lässt sich nun wie folgt schreiben. Die Klasse *Result* dient lediglich dazu, *Permission* und *Channel* gebündelt zurück zu geben.

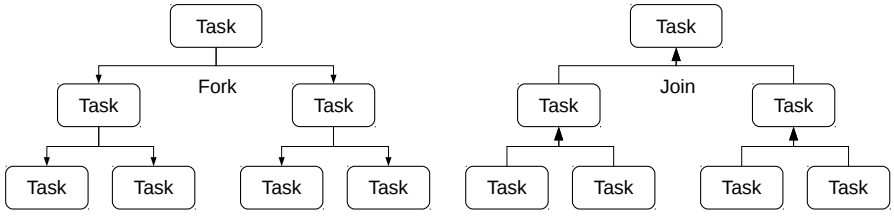
```
1  @GET
2  @Path("watch-channel/{username}/{permission}/{channel}")
3  public boolean watchChannel(@Suspended
4                    AsyncResponse asyncResponse,/*...*/)) {
5     CompletableFuture<Permissions> cPermission =
          us.getUser(userName).thenCompose(user ->
6             ps.getPermission(user.getUserId()));
7
8     CompletableFuture<Channel> cChannel = cs.getChannel(channel);
9
10    CompletableFuture<Result> cResult = cPermission.thenCombine(cChannel,
          (p, c) -> new Result(c, p));
11
12    cResult.thenAccept(result -> asyncResponse.resume(
13       result.getChannel() != null &&
             result.getPermissions().hasPermission(permission)
14    ));
15
16    asyncResponse.setTimeout(500, TimeUnit.MILLISECONDS);
17 }
```

Die drei Services geben nun jeweils ein *CompletableFuture* zurück, mittels *thenCombine*, `thenCompose` und *thenAccept* werden diese zusammen gefügt um am Ende alle Ergebnisse richtig zusammen zu fügen. Zur Sicherheit wir noch ein Timeout von 500 Millisekunden gesetzt um zu vermeiden, dass unnötig lang auf einen nicht reagierenden Service gewartet wird. Die Funktion ist nach der 16ten Zeile beendet, der Thread wird nicht blockiert und kann anderweitig genutzt werden.
Somit ist das Ziel erreicht, durch asynchrone Programmierung einen Thread nicht unnötig zu blockieren.

4.3 ForkJoinPool

ForkJoinPool ist die Grundlage reaktiver Programmierung auf der JVM. Wie die Tasks mit Hilfe des *CompletableFuture*$<T>$ verkettet werden können, wurde bereits erläutert. Es stellt sich nun noch die Frage, wo und von wem werden diese Tasks ausgeführt und was passiert, wenn einer dieser Tasks seine Aufgabe wiederum aufteilen möchte(wie dies bei Aktoren geschehen wird). Der *ForkJoinPool*

ist eine Weiterentwicklung des *ThreadPoolExecuter* [10], dieser arbeitet ähnlich, er besitzt auch einen internen ThreadPool hat jedoch eine gemeinsame Eingangs-Queue, hier entsteht durch den Synchronisationsaufwand ein Flaschenhals. Der *ForkJoinPool* implementiert das Fork/Join-Framework, welches bereits 2000 von Doug Lea vorgestellt wurde[Lea00].
Mit dem *ForkJoinPool* kann ein Task sich aufteilen, die nun erzeugten Subtasks können dann parallel in verschiedenen Threads abgearbeitet werden. Dies macht erst ab einer gewissen Größe des Tasks Sinn, da das erzeugen neuer Tasks natürlich auch aufwendig ist.
Wenn sich ein Task also aufgeteilt hat, wartet dieser so lange, bis seine Subtasks abgearbeitet sind um die Ergebnisse einzusammeln.



(a) Aufsalten in Teilaufgaben (b) Zusammenführen der Ergenisse

Abbildung 6. Funktionsprinzip von Fork/Join

Bei der Erzeugung muss dem *ForkJoinPool* die Anzahl der Threads übergeben werden, die er nutzen soll.

```
1     ForkJoinPool forkJoinPool = new ForkJoinPool(4);
```

Diesem können zwei arten von Aufgaben übergeben werden, zum einen Aufgaben, die kein Ergebnis zurück liefern, dieser werden durch *RecursiveAction* dargestellt und Aufgaben, die ein Ergebnis zurück liefern, diese werden durch *RecursiveTask* dargestellt. Im nachfolgenden Beispiel[11] wird das Aufteilen einer Aufgabe in Teilaufgaben umgesetzt.

```
1 public class MyRecursiveAction extends RecursiveAction {
2
3   private long workLoad = 0;
```

[10] Abgerufen am 1.6.17: `https://www.heise.de/developer/artikel/Das-Fork-Join-Framework-in-Java-7-1755690.html`
[11] Abgerufen am 1.6.17 `http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html`

```
4
5     public MyRecursiveAction(long workLoad) {
6        this.workLoad = workLoad;
7     }
8
9     @Override
10    protected void compute() {
11
12       //if work is above threshold, break tasks up into smaller tasks
13       if(this.workLoad > 16) {
14          System.out.println("Splitting workLoad : " + this.workLoad);
15
16          List<MyRecursiveAction> subtasks =
17          new ArrayList<MyRecursiveAction>();
18
19          subtasks.addAll(createSubtasks());
20
21          for(RecursiveAction subtask : subtasks){
22             subtask.fork();
23          }
24       } else {
25          System.out.println("Doing workLoad myself: " + this.workLoad);
26       }
27    }
28
29    private List<MyRecursiveAction> createSubtasks() {
30       List<MyRecursiveAction> subtasks =
31       new ArrayList<MyRecursiveAction>();
32
33       MyRecursiveAction subtask1 = new MyRecursiveAction(this.workLoad /
             2);
34       MyRecursiveAction subtask2 = new MyRecursiveAction(this.workLoad /
             2);
35
36       subtasks.add(subtask1);
37       subtasks.add(subtask2);
38
39       return subtasks;
40    }
41
42 }
```

Im Beispiel wird eine *RecursiveAction* erstellt, welche ihren *workload* so lange aufteilt, bist er unter eine bestimmte Grenze fällt und damit klein genug ist um von einem Task alleine abgearbeitet zu werden. Wie auch bei Aktoren geben diese Tasks keinen Rückgabewert zurück. Auf diesen müsse sonst mittels *RecursiveAction::join()* gewartet werden. Aktoren kommunizieren rein Asynchron über Nachrichten.
RecursiveAction muss dann einmalig von außen an den *ForkJoinPool* überge-

ben werden und wird von da an rein innerhalb abgearbeitet, ein Blockieren des aufrufenden Threads oder ein Organisationsaufwand(wie es bei der Verwendung von Futures nötig wäre) entfällt.

```
1 MyRecursiveAction myRecursiveAction = new MyRecursiveAction(24);
2 forkJoinPool.invoke(myRecursiveAction);
```

Des weiteren hat der *ForkJoinPool* im Gegensatz zum *ThreadPoolExecuter* nicht nur eine gemeinsame Eingangs-Queue, sondern auch pro Thread eine eigene Queue. Der Thread arbeitet zunächst diese ab, ist seine eigene Queue abgearbeitet, so stiehlt er sich zunächste Tasks aus den Queues von anderen Threads bevor er aus der gemeinsamen Eingangs-Queue einen Task übernimmt. Dieser *Work-Stealing-Alogrithmus* ist in Abbildung 7 dargestellt. Dies bietet zwei we-

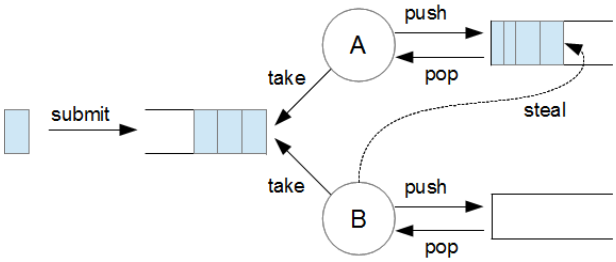


Abbildung 7. Ein ForkJoinPool mit zwei Worker Threads A und B. Neben der Eingangs-Queue existiert zusätzlich eine weitere interne Task Queue pro Thread. Jeder Thread arbeitet zunächst die eigene Queue ab, bevor er sich einer anderen zuwendet

sentliche Vorteile, zum einen nutzen die Threads so lange wie möglich ihre eigene Queue, zum anderen nehmen sie sich gegenseitig Arbeit ab, sofern sie die Kapazität hierzu besitzen.
Um den zusätzlichen Aufwand durch die Queues und das Work-Stealing zu minimieren wurde der *JoinForkPool* stark auf Geschwindigkeit optimiert. Zwei wesentliche Optimierungen sind:

- Die lokalen Queues verwenden eine Deque(double-ended queue), welche das effiziente Einfügen und Entfernen an beiden Enden der Queue erlaubt. Der Worker Thread arbeite nur an einem Ende der Queue indem er seine Tasks auf die Spitze des Stapels legt und diese von dort nimmt(da gerade erzeugte Tasks vermutlich enger mit dem aktuellen Task zu tun haben, als mit einem Task, der fünf Plätze weiter unten im Stapel liegt). Andere Worker Threads, die sich Tasks stehlen wollen, arbeiten wiederum mit dem unteren Ende der Queue. So entstehen Konkurrenzsituationen nur zwischen zwei Worker Threads, die beide auf der Suche nach Arbeit sind.

– Wenn ein Worker Thread keine Tasks stehlen kann, so würde dieser durch das dauerhafte suchen nach Arbeit unnötig Rechenzeit verschwenden. Daher gehen untätige Worker Threads nach einer bestimmten Zeit in einen Ruhezustand und werden bei Bedarf durch das Framework wieder geweckt.

4.4 Reaktive Systemen vs. Reaktive Programmierung

Reaktive Systeme zu entwickeln geschieht auf Ebene der Architektur, *Reaktive Programmierung* hingegen ist ein Werkzeug um die einzelnen Komponenten einer solchen Architektur umzusetzen. Beides ergänzt sich hervorragend, bedingt sich allerdings nicht.

4.5 Umsetzung in Akka

Akka ist ein Toolkit von Lightbend welches es ermöglicht, hochgradig parallele, verteilte, resiliente und nachrichtengetriebene Anwendungen in Java und Scala zu entwickeln.

4.5.1 Aktoren

Aktoren sind Objekte welche Zustand und Verhalten kapseln, diese kommunizieren ausschließlich über Nachrichten, welche in ihrem sogenannten Posteingang ankommen. Somit stellen Aktoren eine sehr strikte Form des objekt-orientierten Ansatzes dar, jedoch kann man Aktoren weniger als Klassen, mehr als Menschen mit Smartphones betrachten. Diese sind von einander unabhängig, können sich aber untereinander Nachrichten senden. Auf eine Antwort wird nicht dauerhaft gewartet, vielmehr können in der Wartezeit andere Dinge getan werden, bis eine Antwort eintritt. Jeder Aktor ist einem `ActorSystem` zugeordnet, dies existiert in der Regel einmal pro logischer Anwendung und belegt meist mehrere Threads auf denen die Aktoren verteilt werden. Die Aktoren werden hierarchisch organisiert und können einander Teilaufgaben zuweisen. Die Hierarchie legt ebenfalls fest, wie Fehler behandelt werden. Jeder Aktor hat genau einen Vorgesetzt(englisch *supervisor*) und wurde von diesem erzeugt um Teilaufgaben abzuarbeiten.
Der Kerngedanken von Aktorsystemen ist, Aufgaben so weit in Teilaufgaben zu zerlegen, bis diese an einem Stück erledigt werden, so wird die Aufgabe klar strukturiert und die Zuständigkeit eines einzelnen Aktors klar geregelt. Dieser Ansatz macht es einfacher für einen Aktor, sein Verhalten bei Fehler und Erfolg zu definieren. Tritt eine Situation auf, mit der ein Aktor nicht umgehen kann, meldet er dies an seinen Vorgesetzten, damit dieser den Fehler beheben kann, oder falls nicht, den Fehler wiederum an seinen Vorgesetzten weiter gibt, bis ein Aktor erreicht ist, der den Fehler kompetent behandeln kann. Im einfachsten Fall, kann ein neuer Aktor gestartet werden, der die Teilaufgabe abarbeitet, im schlimmsten Fall muss der Fehler bis zum Benutzer propagiert werden.
Um festzulegen welcher Aktor welchen andern überwachen soll, gibt es einige Faustregeln.[12]

[12] Abgerufen am 26.5.17:`http://doc.akka.io/docs/akka/current/java/general/actor-systems.html`

- Wenn ein Aktor Teilaufgaben an einen anderen auslagert, so soll dieser diesen überwachen, da der übergeordnete Aktor am besten weiß, wie mit einem Fehler umzugehen ist.
- Aktoren sollten auf keinen Fall blockierenden Programmcode enthalten. Vielmehr soll ein Aktor Anfragen an andere Aktoren stellen und Anfragen beantworten, die an ihn gestellt wurden(Nachrichtenorientiert). Wie mit blockierendem Programmcode umzugehen ist, wird später behandelt.
- Wenn ein Aktor wichtige Daten enthält, welche nicht verloren gehen dürfen, soll er fehleranfällige Teilaufgaben an andere Aktoren auslagern, da ein Fehler dieser gekapselt ist und den übergeordneten Aktor nicht zum Absturz bringen kann.
- Zwischen Aktoren sollen nur unveränderliche Objekte übergeben werden. Sobald der innere Zustand eines Aktors über veränderliche Objekte nach Außen getragen wird, gehen die Vorteile des Aktormodell verloren und es müssen die Probleme normaler paralleler Programmierung bewältigt werden(Sperren von Resourcen bei parallelem Zugriff). Ebenfalls darf auch kein Lambda an einen Aktor gesendet werden, da dies wieder die Grenzen zwischen Aktoren aufweichen würde.

Wie bereits erwähnt, darf in einem Aktor kein blockierender Programmcode stehen, jedoch gibt es Fälle, in denen dieser unumgänglich ist. Das bekannteste Beispiel für solchen Programmcode sind Treiber relationaler Datenbank, nicht alle bieten eine reaktive Schnittstelle an. Bei NoSQL-Datenbanken sind reaktive Treiber meist vorhanden. Der Grund für das blockierende Verhalten ist zu einem großen Teil (Netzwerk) I/O. Diesen blockierenden Aufruf einfach in ein `Future` zu verpacken ist zwar verlockend, führt aber in Produktion schnell zu einem Flaschenhals da hierdurch eine große Zahl an Threads blockiert werden.
Es gibt verschiedenen Ansätze mit solchen blockierenden Aufrufen umzugehen. Nutzt man `Future`, so muss sichergestellt werden, dass die Anzahl der Threads begrenzt ist, da dies unter Last sonst schnell eine große Anzahl Threads generiert, welche das System blockieren. Ein weiterer Ansatz ist der, den blockierenden Aufruf speziell in einen separaten Aktor auszulagern, der einem eigenen Threadpool zugeordnet ist, um wieder nicht das System zu überlasten.

Um seine Aufgabe zu erfüllen kann ein Aktor die Anzahl seiner untergeordneten Aktoren dynamisch erhöhen und reduzieren, wie es die aktuelle Auslastung erfordert.

Überwachung und Monitoring Wie bereits beschrieben bilden die Aktoren in einem Aktorensystem eine Hierarchie, welche für die Fehlerbehandlung verantwortlich ist. Tritt in einem Aktor ein Fehler auf, so unterbricht er, geht in einen Fehlerzustand und meldet dies an seinen Vorgesetzten. Abhängig von der Aufgabe des Aktors und der Art des Fehlers kann dieser nun zwischen vier Optionen wählen:

1. Der Aktor soll fortfahren und behält somit seinen internen Zustand.
2. Der Aktor soll neu gestartet werden und verliert somit seinen internen Zustand.

3. Der Aktor soll dauerhaft gestoppt werden.
4. Den Fehler nach oben weitergeben, also selbst in einen Fehlerzustand gehen.

Option 4 zeigt, dass es wichtig ist, jeden Aktor im Kontext der Hierarchie zu sehen, da jeder Aktor einen übergeordneten Aktor besitzt kann ein Fehler, den er nicht selbst beheben kann, immer nach oben eskaliert werden. Nachfolgend ist solche eine `SupervisorStrategy` als Programmcode dargestellt. Es ist gut zu erkennen, wie hier auf verschiedene Fehler unterschiedlich reagiert wird. Tritt ein Fehler auf, der nicht auf eine der ersten drei Reaktionen passt, wird der Fehler nach oben in der Hierarchie eskaliert.

```
1  private static SupervisorStrategy strategy =
2    new OneForOneStrategy(10, Duration.create("1 minute"),
         DeciderBuilder.
3      match(ArithmeticException.class, e -> resume()).
4      match(NullPointerException.class, e -> restart()).
5      match(IllegalArgumentException.class, e -> stop()).
6      matchAny(o -> escalate()).build());
7
8  @Override
9  public SupervisorStrategy supervisorStrategy() {
10   return strategy;
11 }
```

Da jeder Aktor nur von einem anderen Aktor erstellt und überwacht werden kann, stellt sich natürlich die Frage, wo der oberste Aktor her kommt. Dieser wird von der Bibliothek selbst erzeugt und zur Verfügung gestellt. Dadurch ist sichergestellt, dass die Aktoren nur in einer Hierarchie erstellt werden können, es ist zudem nicht möglich, einen Kind-Aktor einem anderen Aktor als sich selbst unterzuordnen. Dieser könnte dann für den neuen Kind-Aktor keine passende Fehlerstrategie haben beziehungsweise nicht wissen, was er mit diesem tun soll.

Ein `ActorSystem` erzeugt mindestens drei Aktoren: /, /user und /system. Alle Aktoren, welche der Benutzer mittels `system.actorOf()` erzeugt sind Kinder des /user-Aktors. Dieser ist somit der oberste Aktor. Geht dieser in einen Fehlerzustand, beendet der Root-Aktor(/) das komplette `ActorSystem`.

Ortstransparenz Im vorherigen Abschnitt wurde bereits erwähnt, dass Aktoren streng hierarchisch organisiert sind. Jeder Aktor ist über eine Aktor-Referenz zu erreichen, dies stellt die Ortstransparenz sicher. Diese Aktor-Referenz ist ein loser Verweis auf einen Aktor, sie kann zwischen Aktoren übergeben werden, auch über Systemgrenzen hinweg. Die Referenz wird auch durch den Neustart eines Aktors nicht ungültig. Über diese Referenz können dem Aktor Nachrichten gesendet werden, sein interner Zustand bleibt hierdurch jedoch stets verborgen.

Dies ermöglicht es, als Standard ein verteiltes System anzunehmen welches im Spezialfall in nur einer JVM läuft, anstatt von einem lokalen System auf ein verteiltes System zu erweitern, was in der Regel große Änderungen in der Architektur erfordert. Dies hat zur Folge, dass alle Nachrichten, welche gesendet werden, serialisierbar sein müssen.
Die Ortstransparenz in Akka geht so weit, dass nahezu keine API für die Ortung von Aktoren existiert. Wie sich ein System auf verschiedenen Knoten verteilt wird alleine über die Konfiguration festgelegt, hier wird entschieden, welcher Aktor-Teilbaum auf welchem Knoten läuft. Durch das verschieben von Teilbäumen bleiben untergeordnete Aktoren automatisch in der nähe ihres übergeordneten Aktors und die Aufrufzeit somit kurz.

4.5.2 Dispatcher

Das Herzstück von Akka ist der *MessageDispatcher*, dieser ist dafür zuständig, dass die einzelnen Aktoren ausgeführt werden. Jedes *ActorSystem* hat einen solchen Dispatcher. Um die Aktoren auszuführen verfügt jeder Dispatcher über einen *ExecutionContext*. Als Standard hat jedes *ActorSystem* einen Dispatcher mit einem *ForkJoinPool* als *ExecutionContext*, wie dieser Arbeitet wird im nächsten Kapitel erläutert.

Für diesen Dispatcher und *ExecutionContext* können Einstellungen vorgenommen werden. Für den *ExecutionContext* können, abhängig von der konkreten Implementierung verschiedene Parameter konfiguriert werden, unter anderem:

- Maximale Anzahl Threads
- Minimale Anzahl Threads
- Parallelitätsfaktor, dieser gibt an, wie viele Threads pro Kern genutzt werden dürfen.

Der wichtigste Parameter für den Dispatcher ist der *throughput*, dieser gibt die maximale Anzahl Nachrichten an, die ein Aktor verarbeiten darf, bevor der Thread zu einem anderen Aktor wechselt und diesen seine Nachrichten abarbeiten lässt. Wird dieser Parameter auf 1 gesetzt, ist die Verteilung damit möglichst fair.
Hiermit wird deutlich, ein Aktor ist nicht mit einem Thread zu vergleichen. Die Anzahl der Aktoren auf einem wird in der Praxis um ein Vielfaches höher sein, als die Anzahl der Threads. Es wird im Gegenteil sogar versucht, die Anzahl der Threads möglichst gering zu halten um die Anzahl der Kontextwechsel zu minimieren. Ein sehr spannender Artikel zu einer Beispielimplementierung eines Dispatchers mithilfe des *ForkJoinPool* ist bei Heise Online zu finden[13].

4.6 Performance

Anhand eines kleinen Beispielszenarios soll nun der Unterschied zwischen einer blockierendem und einer nicht Blockierenden Anwendung demonstriert werden.

[13] Abgerufen am 5.6.17: `https://www.heise.de/developer/artikel/Das-Fork-Join-Framework-in-Java-7-1755690.html?artikelseite=4`

Hierfür werden für das Darstellen der Webseite drei Services aufgerufen. Diese haben ein Latenz von 500ms. Der Aufrufende Thread, hier das Frontend, führt diese im ersten Fall synchron aus, und benötigt damit 1500 Millisekunden(Abbildung 8). Bis das Frontend einen Antwort senden kann, muss es auf die

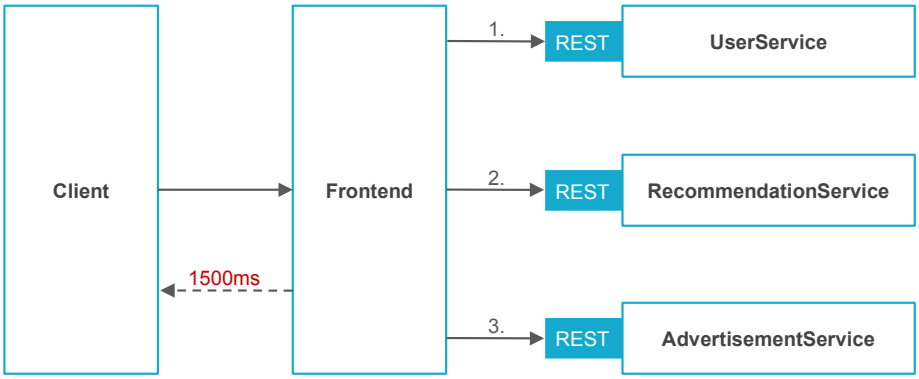


Abbildung 8. Aufbau und Aufrufreihenfolge des Beispielszenarios

Antwort von allen drei Services warten. Wählen wir jetzt einen nicht-blockierenden Ansatz und lagern die Aufrufe der Services in *CompletableFuture*<*T*> aus, so erhalten wir bereits einen Performance Gewinn. Dieser wäre allerdings auch durch rein asynchrone Programmierung in Threads zu erlangen gewesen.
Allerdings benötigt nun jeder Request genau 3 Threads, die jeweils 500ms nur auf eine Antwort warten. Um die effiziente Nutzung der Resourcen hervorzuheben konfigurieren wir unsere Beispielanwendung nun so, dass sie maximal 5 Threads zum abarbeiten der Request nutzen darf.
Für das Testen unserer Anwendung nutzen wir ApacheBench um 500 Requests abzusetzen. Hierbei führen wir immer 100 Stück parallel aus. `ab -c 100 -n 500 localhost:8080/async`

Mit dem nicht blockierenden Aufruf erhalten wir folgendes Ergebnis:

```
1  Benchmarking localhost (be patient)
2  [...]
3  Finished 500 requests
4
5  Server Hostname:      localhost
6  Server Port:          8080
7
8  Concurrency Level:    100
9  Time taken for tests: 5.122 seconds
10 Complete requests:    500
```

Es wurden 500 Requests in 5,122 Sekunden abgearbeitet, vergleichen wir dies nun mit der Leistungsfähigkeit bei blockierenden Threads, so erhalten wir folgendes Ergebnis:

```
Benchmarking localhost (be patient)
[...]
Finished 500 requests

Server Hostname:      localhost
Server Port:          8080

Concurrency Level:    100
Time taken for tests: 250.762 seconds
Complete requests:    500
```

Wir erhalten also ein Ergebnis von 4,17 Minuten. Hier zeigt sich deutlich, wie groß der Unterschied ist, ob Threads blockieren, oder ob Threads anderweitig genutzt werden können. Gerade bei modernen Microservice Architekturen bei der viele Aufrufe an andere Services und Datenbanken erfolgen, kann dadurch ein signifikanter Leistungsgewinn verzeichnet werden.

5 Umsetzung in Lagom

Ein Framework das die oben erläuterten Konzepte umsetzt ist Lagom von Lightbend. Mit der Namensänderung von Typesafe zu Lightbend[14] hat die Firma ihre neue Plattform vorgestellt, welche es ermöglichen soll, reaktive Microservices zu entwickeln. Lagom baut auf existierenden Technologien wie *Akka*, *Play Framework* und *Apache Cassandra* auf und bringt somit gleich einen großen Teil der Infrastruktur mit. Ein konfiguriertes Projekt kann mit Lagom per `sbt runAll` auf jedem Entwickler-System gestartet werden und bringt die benötigten Systeme wie Message-Queue und Datenbank gleich mit.
Lagom nutzt *Akka* und setzt damit stark auf das Aktorenmodell. Auch wenn das Aktorenmodell bereits 1973 in einem Artikel von Carl Hewitt, Peter Bishop und Richard Steiger[Hewi] beschrieben wurde, eignet es sich laut einer Studie von Forrester Research[Hamm16] sehr gut für die Entwicklung paralleler Anwendungen in der Cloud.

Asynchron als Standard In Lagom erfolgt die Kommunikation zwischen Services per Default asynchron. Hierzu kommen Akka Streams zum Einsatz. Für den asynchronen Programmfluss setzt die Java API auf `java.util.concurrent.CompletableFuture` und die Scala API auf `scala.concurrent.Future`.

Technologiefreiheit Lagom setzt nicht voraus, dass alle Services im System auf Lagom basieren. Durch die Verwendung von HTTP für synchrone und Websockets für asynchrone Kommunikation können beliebige Programmiersprachen in das System eingebunden werden.

[14] `https://www.lightbend.com/blog/typesafe-changes-name-to-lightbend`

Persistenz Lagom bevorzugt eine verteilte Datenhaltung. Jeder Service hat sein eigenes Datenbank-Schema. Standardmäßig nutzt Lagom *Event-Sourcing* und *Command Query Responsibility Segregation*(CQRS)[BDMS+13].
Durch den Einsatz von *Event-Sourcing* werden alle Änderungen in der Datenbank als unveränderliche Tatsache(eng. *immutable fact*) hinterlegt. Wird Geld an einem Bankautomat abgehoben, so wird dies in der Datenbank als „10€ abgehoben“ gespeichert. In einem klassischen relationalen Datenbanksystem müsste hierzu zuerst der aktuelle Betrag abgefragt werden, um 100 reduziert und der neu berechnete Kontostand in die Datenbank geschrieben werden. Über diese ganze Zeit müsste sichergestellt werden, dann kein parallel laufender Prozess den Kontostand ebenfalls ändert.

5.1 Entwicklungsumgebung

Lagom setzt auf das Build-Tool sbt[15], welches aus der Scala-Welt kommt. Die Entwicklung ist ebenfalls mit Apache Maven möglich, empfohlen wird aber der Einsatz von sbt. Ein Lagom Projekt kann damit einfach über folgenden Kommandozeilenaufruf erzeugt werden:

```
sbt -Dsbt.version=0.13.15 new https://github.com/lagom/lagom-java.g8
```

Nach der Konfiguration des Projektnamen und anderen Parametern kann die Entwicklungsumgebung mit folgendem `sbt runAll` gestartet werden, die Services sind nun über den *Service Gateway* unter `http://localhost:9000` erreichbar. Hierdurch werden neben den Services auch folgende Dienste gestartet:

Apache Kafka Ist ein *Message Brocker* und für die Verteilung von Nachrichten im System zuständig. Durch den Einsatz von Nachrichten wird eine losere Kopplung der einzelnen Services erreicht.

Apache Cassandra Lagom liefert von Haus aus eine Cassandra Datenbank mit, die mit der Lagom internen Persistenzmechanismen gut zusammen spielt.

Service Locator Um die Ortstransparenz zu erreichen bringt Lagom eine Service Registry und einen Service Gateway mit, somit bleibt es für die konkreten Services verborgen, an welchem Ort sich ein anderer Service befindet.

Jedem Service wird automatisch ein Port zugeordnet über den er erreichbar ist, dies gescheit konsistent über den Namen des Services. Es wird eine Hashwert aus dem Namen generiert und in den Portbereich abgebildet, der Lagom zur Verfügung steht. Standard ist hier 49152 bis 65535. Ist der Port bereits belegt, wird der nächst höhere frei Port verwendet. Bei Bedarf kann der Port für einen Service auch in der Konfiguration festgelegt werden.

5.2 Komponenten

Lagom bündelt verschieden Technologie, einige sind selbst entwickelt, andere sind bekannte Open Source Technologien.

[15] `http://www.scala-sbt.org/`

Wie bereits erwähnt bringt Lagom Apache Kafka und Apache Cassandra mit. Für die *Persistenz*, den *Publish-Subscribe* Mechanismus und das *Clustering* setzt Lagom auf Akka. Es ist zwar möglich Akka API's direkt aufzurufen, jedoch kommt man bei der Benutzung von Lagom nicht mit ihnen in Kontakt. Des weiteren kann das, ebenfalls von Lightbend entwickelte, *Play Framework*[16] für die Umsetzung von Webseiten eingebunden werden, durch seinen ebenfalls reaktiven Charakter eignet sich dieses Framework besonders für den Einsatz in Reaktiven Microservices.

6 Fazit und Ausblick

Es wurde gezeigt, wie Systeme gebaut werden können, die effizient mit Resourcen umgehen und einer hohe Last standhalten. Dies bringt jedoch auch zusätzliche Komplexität in die Architektur und in die Entwicklung. Ob sich dieser Mehraufwand im speziellen Fall lohnt muss abgewogen werden, die Mehrheit der Anwendungen werden wohl vorerst keinen Bedarf sehen, auf ein komplett Reaktives System wechseln zu müssen, die aktuellen Entwicklungen zeigen jedoch, dass der reaktive Ansatz immer populärer wird.

6.1 Anwendungsbereiche

Bei der Recherche ist aufgefallen, dass viele der Konferenztalks über Reaktive Programmierung beziehungsweise Reaktive Systeme von großen Firmen stammen, deren Geschäftsmodell stark von der Fähigkeit zu Skalieren abhängt, hierbei waren unter anderem Sky(Online TV vgl. Netflix)[Bate16], Spotify[17] und selbstverständlich Netflix[18] wo im Microservice Umfeld ohnehin viel Innovation einbringt.
Es ist klar geworden, dass Microservices, gerade in Kombination mit reaktiver Programmierung, eine Technologie sind, mit der eine hohe Skalierung und gute Ausnutzung der Hardware-Resourcen möglich wird. Ein Thread kostet pro Jahr auf einer AWS EC2 Instanz $8, diese Kosten muss man mit dem Mehraufwand bei der Entwicklung gegen rechnen, den reaktive Programmierung mit sich bringt. Spart man hierdurch 10 Threads, wird sich dies kaum lohnen[Bate16]. Verarbeiten die Systeme allerdings tausende von Anfragen pro Sekunde, so können hierdurch signifikant Resourcen und damit Kosten eingespart werden.

6.2 Aktuelle Entwicklungen

6.2.1 Spring 5

Das Spring-Team stellte sich vor circa anderthalb Jahren die Frage, ob eine Integration des Reaktiven-Paradigmas in Spring Sinn ergibt. Dabei spielten einige

[16] `https://www.playframework.com/`
[17] Abgerufen am 24.5.17: `https://www.youtube.com/watch?v=sRODBugm57s`
[18] Abgerufen am 24.5.17: `https://www.youtube.com/watch?v=5GkOVKuG8yY`

Faktoren eine Rolle, welche die Umsetzung als auch das Timing beeinflussten. Spring kommt meist eher im konservativen Unternehmensumfeld zum Einsatz, also in Unternehmen, die neuen Technologietrends eher später aufnehmen. Umso mehr, wenn diese Trends eine größere Konzeptionelle Änderung an der bestehenden Vorgehensweise und Architektur fordern. Jedoch hält auch in diesen Unternehmen der Trend zu Microservices als Architekturparadigma Einzug. Große Monolithen werden in kleinere Systeme aufgeteilt, die miteinander kommunizieren. Da diese Systeme meist verschachtelte Aufrufe mit sich bringen und unter hoher Lastanforderung stehen ist es nötig, dass sie eine möglichst geringe Latenz aufweisen. Zudem müssen nun mehr dieser kleinen Systeme auf der selben Hardware laufen, also zuvor wenige große, dies bedeutet, dass diese Systeme ihre Resourcen nun effizienter nutzen muss.
Ziel war es daher auch, ein Framework zu entwickeln, welches des bisherigen Nutzern erlaubte, möglichst viel Wissen aus dem bisherigen Spring MVC weiter zu verwenden und dennoch nicht zu vernachlässigen, dass dieser neue Ansatz signifikante Änderungen bei der Verwaltung von Resourcen und Transaktionen mit sich bringt. Hieraus ist das Spring WebFlux Framework[19] entstanden. Damit ist nun eine Alternative für das bisherige Spring MVC Framework geschaffen worden. Es ist jedoch kein einfacher Ersatz für Spring MVC, die Entscheidung, auf welchem Framework ein Projekt aufsetzen soll, muss zu Beginn sehr bewusst getroffen werden[Gier17].

6.2.2 Reaktive in Java 9

Auch in Java 9 findet der reaktive Ansatz Einzug. Mit der FlowAPI[20] ist nun ein Standard verfügbar, der sich an den Interfaces von JavaRx, Reactor und einigen anderen reaktiven Erweiterungen für Java orientiert. Mit dem *Flow.Subscriber*$<T>$ Interface lassen sich damit reaktive Datenflüsse aufbauen. Wie auch bei *CompletableFuture*$<T>$ werden die Daten erst verarbeitet, wenn sie vorhanden sind. Dies nennt sich *ReactiveStreams* und ist auch in Akka implementiert und dient in Lagom dazu, große Datenmengen zwischen einzelnen Services zu übertragen.

```
1 public static interface Flow.Subscriber<T> {
2   public void onSubscribe(Flow.Subscription subscription);
3   public void onNext(T item);
4   public void onError(Throwable throwable);
5   public void onComplete();
6 }
```

[19] Abgerufen am 16.5.17: `https://spring.io/blog/2017/02/23/spring-framework-5-0-m5-update`
[20] Abgerufen am 1.6.16: `https://community.oracle.com/docs/DOC-1006738`

Literatur

Bate16. Christopher Batey. Devoxx Poland - The Java developers' guide to asynchronous programming. Poland, 2016. Abgerufen am 24.5.17 `https://www.youtube.com/watch?v=F32XoAPijTo`.

BDMS+13. Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi und Mani Subramanian. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure*. Microsoft patterns & practices. 1st. Auflage, 2013.

Brew00. Eric A. Brewer. Towards Robust Distributed Systems, 2000.

Conw68. Melvin E. Conway. How Do Committees Invent? *Datamation*, April 1968.

Erb12. Benjamin Erb. Concurrent Programming for Scalable Web Architectures. Diploma Thesis, Institute of Distributed Systems, Ulm University, April 2012.

Gier17. Höller Jürgen Paluch Mark Gierke, Oliver. Spring 5 - Frühling für Enterprise Java. *Javamagazin*, Band 5, 2017, S. 40–49.

Hamm16. Rymer John R. Hammond, Jeffrey S. How To Capture The Benefits Of Microservice Design. *Forrester*, 2016.

Hewi.

JaPa13. Ahmad A. Jamshidi, P. und C. Pahl. Cloud migration research: a systematic review. *IEEE Transactions on Cloud Computing*, Band 1, 2013, S. 142–157.

Lea00. Doug Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, New York, NY, USA, 2000. ACM, S. 36–43.

MoDe16. A. Mouat und T. Demmig. *Docker: Software entwickeln und deployen mit Containern*. Dpunkt.Verlag GmbH. 2016.

Newm15. *Microservices: Konzeption und Design*. mitp, [Frechen]. 1. Aufl.. Auflage, 2015.

Nyga07. Michael T. Nygard. *Release it! : design and deploy production-ready software*. The pragmatic programmers. Pragmatic Bookshelf, Raleigh, NC [u.a.]. 2007.

Rote06. A. Rotem-Gal-Oz. Fallacies of distributed computing explained. 2006. Abgerufen am 23.5.17 `http://www.rgoarchitects.com/Files/fallacies.pdf`.

Shar16. Sourabh Sharma. *Mastering microservices with Java*. Community experience distilled. Packt Publ., Birmingham. 2016.

Vern17. Vaughn Vernon. *Domain-Driven Design kompakt*. dpunkt.verlag, Heidelberg. 201704, 1. Auflage. Auflage, 2017.

Wolf17. Eberhard Wolff. *Microservices Ein Überblick*. innoQ. 2017.