Distributed systems

Jakob Klemm

2. Juni 2021

Inhaltsverzeichnis

Abstract: *Distributed systems* sind schwer. Zu planen, zu schreiben, zu verwalten und sich vor zu stellen.
Von menschlichen und praktischen Problemen, bis hin zu physikalischen Gesetzen stehen *distributed systems* viel im Weg. Tatsächlich müssen wir feststellen, dass es unmöglich ist ein perfektes oder Nachteil loses *distributed system* zu bauen. Dazu kommt, dass die Programmierung von solchen Systemen sehr schwer werden kann und die nötigen Programme, Frameworks und Tools meist nicht existieren. Dadurch sind wir in eine Situation gekommen in der wir als einfachsten Weg ein *distributed system* zu schreiben versuchen, keines zu schreiben. Wir arbeiten verzweifelt daran unsere Applikationen frei von jeglichen Daten oder Zuständen zu halten, da wir nicht in der Lage sind diese richtig zu verwalten.
Mit all diesen Nachteilen, Problemen und Unsicherheiten muss man sich die Frage stellen ob es sich wirklich lohnt so viel Zeit und Aufwand in ein *distributed system* zu investieren. Aber wenn richtig implementiert sind die daraus gewonnen Vorteile unglaublich gross und der Verwaltungsaufwand deutlich geringer.
Dazu kommt auch dass es immer einfacher wird *distributed systems* zu schreiben. Durch neue Programmiersprachen und Frameworks sinkt der Aufwand täglich. Natürlich sind diese Tools nicht in der Lage die physikalischen Gesetze zu umgehen oder magische Lösungen zu liefern. Aber sie machen es schnell und einfach innerhalb dieser Richtlinien zu operieren und sichtbare Resultate zu liefern.
Leider gibt es auch nach der Entscheidung, ein *distributed system* zu bauen noch viele Möglichkeiten und Probleme. Meist gibt es keine *richtige* Antwort und man muss schlussendlich zwischen zwei weniger guten Varianten (Beispielsweise *peer-to-peer* oder *fully connected mesh*) auswählen. Und trotzdem sind die Vorteile die von einem *distributed system* in der richtigen Situation kommen äusserst gross und vorteilhaft für alle involvierten. Von den Programmieren, über die Arbeiter die das System bedienen und verwalten müssen, bis hin zu den Endnutzern die meist geringere Latenzzeiten und konstante Daten geliefert bekommen.

Teil I

Einleitung

In diesem ersten Abschnitt wollen wir die Geschichte der Computer & Prozessoren anschauen. Dabei soll der Fokus auf den Abschnitten der Geschichte ligen, die später für distributed systems relevant wurden.
Befor wir aber damit anfangen müssen wir uns fragen, was *distributed systems* eigentlich sind?
Einfach gesagt lässt sich ein solches system als eine Sammlung von Geräten oder Maschinen definieren, die gemeinsam kommunizieren und Berechnungen durchführen. Meist geht es dabei nicht um eine einzige Rechnung, die über verschiedene Stationen aufgeteilt wird, sondern um eine vielzahl von kleineren Berechnungen die aber alle von einander abhängig sind und miteinander kommunizieren. Ein gutes Beispiel dafür sind Webserver, die viele kleine Requests verarbeiten müssen und gleichzeitig mit grossen Datenmengen konfrontiert werden.
Man sagen, dass das Verlangen für *distributed systems* aus dem Problem heraus entstand, dass ein einzelner Computer nicht mehr genug war, um die benötigten Berechnungen durchzuführen oder die Verantwortung für ein einzelnes System nicht einem einzelnen Gerät überlassen werden konnte. Aber es kommen viele Probleme auf sobald man Berechnungen, und daher auch Daten, auf mehreren Geräten verteilt. Vor allem wenn es um das Verwalten von Zuständen und Daten geht, existieren haufenweise Probleme und Unsicherheiten.
Da diese Themen oftmals sehr Abstrakt wirken können, ist es möglicherweise hilfreich, ein Beispiel dazu zu analysieren. Dabei muss man sich bewusst machen, dass wir täglich und dauerhaft mit solchen Systemen interagieren. Das häufigste Beispiel ist wahrscheinlich das Internet selbst. Dieses verwendet `IP` und `DNS`, sowie `TCP` um den Zugang zu Daten und Ressourcen zu ermöglichen. Aber da das Internet ein Zusammenschluss von verschiedenen Servern und Firmen ist, gibt es eine vielzahl von Mechanismen zu Rechten und Zugangängen die wir in anderen Systemen nicht finden werden. Trotzdem kann man das Internet selbst als einen einzigen Zusammenschluss von Prozessen sehen. Leider ist das Internet nicht das effizienteste System, wenn es um den Zugriff auf Daten geht.
Daher ist es eher selten, dass eine Firma oder Webseite versucht ihre eigene Infrastruktur nach der des Internets zu modellieren. Trotzdem können viele der Prinzipien, die für das Internet funktionieren, für allgemeine verteilte Systeme extrahiert werden.
Aber nun zu einigen historischen Hintergründen:
Im Jahre 1946 kam der *erste* Computer ans Netz. Auch wenn man den `ENIAC` heute kaum noch als Computer erkennen würde, basiert er dieser fundamental trotzdem auf den gleichen Prinzipien wie heutige Computer. Der `ENIAC`

wurde speziell für die Berechnung von Raketen- und Projektil-Flugbahnen gebaut. Er war in der Lage 357 Berechnungen pro Sekunde durchzuführen. In späteren Iterationen wurden die Elektronenröhren des ENIAC dann durch Transistoren ersetzt, wodurch die Kapazität der Maschine bei geringerer Grösse drastisch erhöht wurde.
Eine der nächsten wichtigen Verbesserungen kam dann für die Apollo-Missionen. Da alle der Computer zur steuerung der Rakete unter enormen Gewichtsbeschränkungen standen, musste die Kapazität der Computer erhöht werden, ohne mehr Transistoren hinzuzufügen. Mit dem Apollo-Computer wurden zum ersten mal Prozesse mit Prioritäten berechnet. So war es möglich, die wichtigsten Aufgaben, wie das Steuern der Rakete unter allen Umständen durch zu führen, während weniger wichtige Aufgaben wie das Senden von Telemetriedaten pausiert werden konnte.
Mit neuen Verfahren zur Berechnung, sowie besserer Hardware wurden Computer immer schneller und besser.

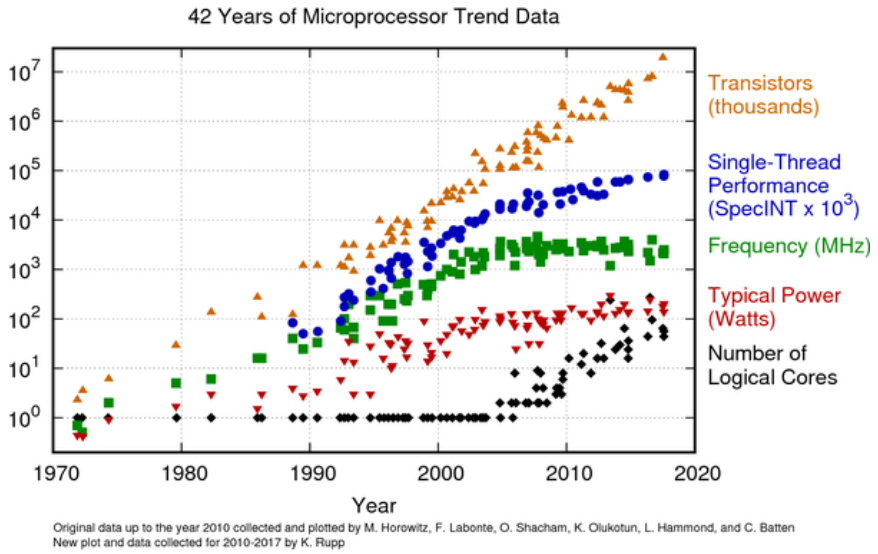
Kapitel 1

Moores Law

Wir haben bereits gesehen, dass von ENIAC zu Apollo grosse Fortschritte in Sachen Transistorarchitektur, Bauverfahren und Leistung gemacht wurden. Dabei sah man eine Zunahme der Leistung um ein vielfaches, wobei die Grösse dauerhaft abnahm.
Dieses Konzept wurde bereits 1965 von Gordon Moore zu Papier gebracht, wobei er versuchte, Mathematische Regeln zu verwenden. Was aus seinem Wissen entstand ist heute allgemein als moores law bekannt.

Moores law besagt, dass sich die Komplexität integrierter Schaltkreise regelmässig verdoppelt, meist in 12 Monaten.
Unter Komplexität ist die Anzahl der Schaltkreiskomponenten auf einem integrierten Schaltkreis gemeint. Gelegentlich ist auch von einer Verdoppelung der Integrationsdichte die Rede, also der Anzahl an Transistoren pro Flächeneinheit.
Da sich dieses Gesetz auch in der Praxis sehr einfach sehen lässt und mit mehr Transistoren auch meist mehr Leistung kamen, hatten sich Kunden an diese Verdopplung gewöhnt. Zu dieser Zeit war eine der besten Möglichkeiten, mehr Leistung aus seinem Programm zu bekommen, ein Jahr zu waren. Nach dieser Zeit lief das Programm meist etwa doppelt so schnell.
Nur leider kamen andere physikalische Faktoren ins Spiel, die dieser konstanten Verdopplung ein Ende bereiteten. Zum einen wurde es immer schwerer, die Grösse der Bauteile zu senken. Aktuell geht man von etwa 7nm pro Transistor aus, aber selbst Firmen wie Intel haben immer noch Probleme diese Bauverfahren zu implementieren. Zwar wird bereits an 3nm geforscht, aber der Fortschritt wird dann oder kurz danach ein Ende haben, da man die physikalischen Limits erreicht und inzwischen schon in einzelnen Atomen messen

und rechnen muss.
Dazu kommt noch das Problem der Leistung, beziehungsweise der Wärme. Ein einzelner Prozessor kann nur eine limitierte Menge an Wärme produzieren, sonst ist es schlicht unmöglich ihn kühl zu halten. Auch wenn es wenige Ausnahmen gibt, haben die meisten modernen Prozessoren nicht mehr als `100` Watt Abwärme. Da die Fläche eines Prozessors so gut wie immer limitiert ist, sind Kühler nur in der Lage eine begrenzte Menge Wärme abzutransportieren. Es ist also unwahrscheinlich, dass wir in naher Zukunft in diesem Bereich drastische Fortschritte sehen werden.
Auch die Frequenz, also die Berechnungen pro Sekunde, sind eher stagniert. Auch hier muss man wieder auf physikalische Faktoren verweisen, die es sehr schwer machen, die Frequenz zu erhöhen. Einfach gesagt kann man es wieder auf ein Hitze Problem zurück führen.
Aber die Welt hatte sich an diese Verdopplung der Leistung gewöhnt. Man konnte nicht einfach ankündigen, dass keine Verbesserungen mehr kommen würden und man mit der aktuell vorhandenen Leistung auskommen müsste. Also entschied man sich, anstelle von besseren Prozessoren einfach mehr Prozessoren zu bauen. Man nannte die einzelnen Prozessoren auf einem Sockel Kerne und begann zwei, vier und acht Kern-Prozessoren zu produzieren. Über die Jahre visualisiert erhält man dann diese Grafik. [7]



Die verschiedenen Eigenschaften der Prozessoren sind über die Jahre gezeichnet. Am deutlichsten kann man die Abflachung verschiedener Eigenschaften ab `2000` und den gleichzeitigen Anstieg der Anzahl Kerne erkennen. Damit

kommt aber ein neues Problem auf:
Ein Programm wird nicht doppelt so schnell, wenn man es auf doppelt so vielen Kernen laufen lässt. Programme müssen speziell geschrieben werden, um von mehreren Kernen profitieren zu können. Tatsächlich können schon alleine Multi-Core programme als ein vollständiges *distributed system* gesehen werden.

Kapitel 2

Multi-Cores

Wieso sind Programme nun also nicht schneller, wenn sie auf mehreren Kernen laufen?
Die meisten *klassischen* Programme bestehen aus einer Reihe von Instruktionen. Diese werden nacheinander ausgeführt um am Ende ein Resultat zu erhalten. Daher spielt es keine Rolle, wie schnell jede einzelne Berechnung gemacht wird, solange die Reihenfolge bestehen bleibt. Aber sobald es um Multi-Core Prozessoren geht, können Berechnungen gleichzeitig stattfinden, womit aber die meisten Programme nicht klar kommen.
Dadurch entsteht eine Unmenge an Problemen und Unsicherheiten. Beispielsweise gibt es `Race-Conditions`.

2.1 Race-Condition

Da bei einem Computer mit mehreren Kernen nicht alles mehrfach vorkommt, gibt es Komponenten, beispielsweise der Speicher, auf die alle Kerne zugreifen. Nehmen wir nun als Beispiel an, dass im Speicher ein Zähler also ein einfacher Zahlenwert, gespeichert ist. Dazu haben wir noch zwei Kerne, die eine komplizierte Berechnung durchführen. Dabei spielt es keine Rolle ob dies die gleiche Rechnung ist oder nicht. Jedes mal wenn sie ein Resultat finden soll der Wert im Speicher um 1 erhöht werden.
Dabei ist es wichtig zu verstehen, wie Computer einen Wert tatsächlich verändern können. Dafür kopieren sie den Wert in ihren Arbeitsspeicher, führen eine Berechnung (Erhöhung um `1`) durch und schreiben den neuen Wert an die Stelle des alten, womit sie diesen überschreiben.
Nehmen wir nun an, wir haben `42` im Speicher. Zur *gleichen* Zeit finden dann beide Kerne ein Resultat, kopieren `42` zu sich, ändern den Wert und

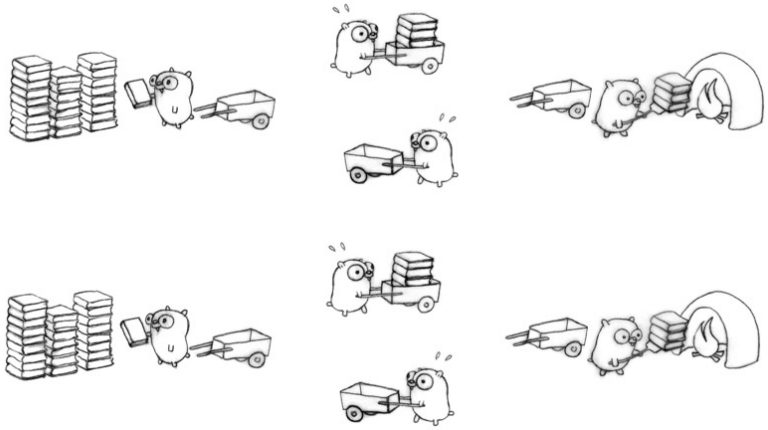
schreiben es wieder in den Speicher. Da wir zwei neue Resultate haben, erwarten wir als neuen Wert `44`. Aber die beiden Prozesse haben zwei mal `42` kopiert und dadurch zwei mal `43` geschrieben.
Natürlich gibt es möglichkeiten, dagegen vorzugehen, beispielsweise mit `Locks`, aber es ist momentan nur wichtig, dass die Problematik verstanden ist.

2.2 Parallelisierung & Nebenläufig

Natürlich ist es nicht nur schlecht. Mehrere Kerne erlauben es, mehrere Berechnungen gleichzeitig durchzuführen. Dabei wird meist zwischen zwei Kategorien unterschieden:

1. Parallelisierung: Wenn ein Prozess parallel ausgeführt wird, bedeutet dies, dass die mehr oder weniger gleiche Berechnung zur etwa gleichen Zeit auf mehreren Kernen ausgeführt wird. Als Beispiel kann man sich ein Programm vorstellen, dass ein bestimmtes Wort in verschiedenen Dateien sucht. Man kann also für jede Datei einen Prozess starten, der auf einem eigenen Kern läuft. Auch wenn es gerine Ineffizienzen gibt, kann man etwa sagen, dass eine solche Berechnung auf `n` Kernen `n-Mal` schneller läuft.

2. Nebenläufigkeit: Aber manchmal ist es nicht möglich, oder nicht effizient, ein Programm so aufzuteilen. Da kommt dann Nebenläufigkeit oder auch Concurrency ins Spiel. Dabei macht nicht jeder Prozess das Gleiche. So kann man sich Nebenläufigkeit gut mit dieser Grafik [16] der `Golang` Programmiersprache vorstellen:
   (Wieso die Entwickler der Sprache als Beispiel das Verbrennen von Büchern gewählt haben bleibt weiterhin unklar, aber es sagt einiges über die Community der Sprache aus.)



In diesem Beispiel geht es darum Bücher zu verbrennen. Um dies möglichst effizient zu erledigen, gibt es einen Prozess (Den wir uns als einen Kern eines Prozessors vorstellen können) der die Bücher aus dem Regal nimmt, einen der die Bücher zum Ofen bringt, einer der sie verbrennt und einen letzten, der die leeren Wagen wieder zum Regal zurück bringt. Um dann den gesamten Prozess noch schneller beenden zu können, lassen wir dieses System aus Nebenläufigen Prozessen zwei mal in Parallel laufen, wobei die beiden Blöcke je das Gleiche erledigen. Der Vorteil an diesem System ist es, dass jeder Prozess unabhängig von den Anderen seine Aufgaben erledigen kann. Wenn also der Ofen überhitzt, können weiterhin Bücher transportiert werden, ohne dass das gesamte System zusammenbricht. Auch wenn es nur bedingt für Computer gilt, kann man sich auch vorstellen, dass es dieses Modell sich auf einzelne Aufgaben zu spezialisieren. Der Prozess für den Ofen kann daher seine volle Aufmerksamkeit dem Ofen widmen, und muss sich keine Gedanken über die Bücherregale machen.
In einem praktischen Beispiel ist dies hilfreich, da es dem Prozess für den Ofen egal sein kann, ob die Bücher aus einem Regal oder von einem anderen Ort her kommen, da er sich nur um das verbrennen kümmern muss.
Natürlich kommen auch damit neue Probleme auf. Man muss sich um den Fall kümmern, dass einer der Prozesse *idle* ist (Nichts zu tun hat). Oftmals schalten sich Prozesse in einer solchen Situation gerne ab, allerdings wäre es für unser Beispiel natürlich unpassend wenn der Ofen plötzlich abgeschalten wird, nur weil eine neue Ladung Bücher geholt werden muss. Natürlich ist es wichtig am Ende, nachdem alle Bücher verbrannt wurden, keine der Prozesse zu vergessen oder unnötig lan-

ge aktiv zu lassen. Meist drückt man dieses Verhalten im Code durch ein Wartesystem aus, wobei der *Hauptprozess* auf alle anderen wartet. Allerdings funktioniert so etwas nur, wenn die Prozess auch in der Lage sind zu signalisieren dass sie ihre Arbeit abgeschlossen haben. Sollte einer Abstürzen oder in einen unendlichen Loop geraten kann es zu Komplikationen kommen. Wir werden später noch `Elixir's` Supervisors anschauen, die sich mit diesen Problemen befassen.

Kapitel 3

Moderne Infrastruktur

Leider geht es inzwischen nicht nur um mehrere Kerne und Prozessoren, sondern auch um Netzwerke und besonders das Internet. Anstelle von einzelnen Prozessen die direkt nebeneinander in einem Prozessor ablaufen, haben wir nun Prozesse die über grosse Distanzen (sowohl physikalische Distanzen als auch Zeitliche) voneinander getrennt sind.
Da viele der Probleme mit denen man sich heute im Zusammenhang mit *distributed systems* auseinandersetzen muss mit dem Internet in Verbindung stehen ist es wichtig, dass dieses Thema noch genauer besprochen wird.
Dabei kann man *distributed systems* und das Internet in zwei Kategorien aufteilen:

3.1 Internet

Wie oben bereits angesprochen ist das Internet selbst ein *distributed system*, auch wenn wir selten darüber nachdenken, weil wir es nie als ein einzelnes, vollständiges System ansehen. Das Internet besteht aus millionen von Servern, die tausenden von Firmen gehören. Daher ist es schwer sich das Internet als *ein* System vorzustellen. Aber tatsächlich folgen diese millionen von Servern den selben Standards und unterstützen dieselben Protokolle. Aber das Internet ist nicht besonders Effizient, wenn es um das Speichern und Erreichen von Daten geht.

1. IP & DNS: Jeder hat schon mal eine Domain in seinen Browser eingegeben und wir stellen uns vor, dass das Internet auf Domänen und `DNS` aufbaut. Aber tatsächlich ist `DNS` nur eine Abstrahierung über dem Internet-Protocol. Wahrscheinlich kann man sich auch unter einer `IP-Adresse` noch etwas vorstellen, aber es ist wichtig zu verstehen,

dass jede Domain eigentlich nur ein Schlüssel ist, der auf eine bestimmte Domain zeigt. Da sich aber Menschen diese Adressen nicht gut merken können und diese sich auch ändern verwenden wir meist Domänen. Auch kommt dazu, dass Domänen in keinem logischen Zusammenhang zum Inhalt stehen. Während ein `sha1` Hash den ursprünglichen Wert exakt repräsentiert, sagt uns die Domäne *domain.com* nichts über den Inhalt den man dort finden kann aus.
Mit abstrakten Konzepten wie der Sprache und etwas Logik können wir erraten, dass *domain.com* zu einer Seite für das Kaufen von Domänen gehört, aber für einen Computer ist dieser Schritt unmöglich.
Alles in allem machen diese Probleme, zusammen mit weiteren die noch später angesprochen werden, das Internet eine gute Lösung für Menschen, aber sind insgesamt nicht besonders Effizient, vor allem für Computer.

3.2 Server

Neben dem Internet selbst gibt es noch weitere Probleme, die erst in den letzten Jahren entstanden sind. Durch populäre Seiten wurde es unmöglich, die gesamte Funktionalität einer Seite auf nur einem Server zu implementieren. Zusätzlich ist diese Option auch nicht sonderlich Fehlertolerant. Also ist inzwischen nicht nur das Internet selbst ein *distributed system* sondern die einzelnen Server die es ausmachen. Dabei gibt es zwei häufige Implementierungen oder design Richtugen. Weitere Beispiele zu den beiden werden wir später noch genauer Anschauen, für den moment reicht es die beiden beliebtesten etwas genauer kennen zu lernen.

1. Microservices: Die `Microservice`-Architektur ist inzwischen eine sehr beliebte Option für grössere Projekte und komplizierte Systeme.
   Ähnlich wie man seinen Code in Module, Klassen und Funktionen aufteilt funktionieren auch `Microservices` nach diesem Prinzip. Der einzige Unterschied ist dabei die grösse der einzelnen Komponenten.
   Als sehr gutes Beispiel für eine `Microservices`-Architektur kann man *accounts.google.com* anschauen:
   Der Service läuft unabhängig von den restlichen `Google`-Diensten und erfüllt nur eine Aufgabe: Er kümmert sich um Accounts. Der Dienst ist also eine *kleine* unabhängige Einheit, die von anderen Diensten integriert werden kann. Beispielsweise auf `YouTube` leitet der *Login*-Button auf *accounts.google.com* weiter, genau gleich wie der *Login-Button* auf *Gmail* oder ein beliebiger anderer Google-Dienst. Auch

wenn es natürlich eine unglaubliche Vereinfachung darstellt, kann man sich Seiten wie `YouTube` als eine Reihe von solchen Diensten vorstellen. Einer kümmert sich nur um Accounts, alle anderen Dienste kommunizieren dann einfach mit dem dedizierten Accounts-Dienst für alle Infos und Aktionen rund um Accounts. Ein weiterer kümmert sich nur um Videos, ein dritter nur um Vorschläge für die Nutzer. Dadurch ist es zum einen möglich, die einzelnen Dienste unabhängig zu entwickeln. Ein wichtiger limitierender Faktor für viele Projekte dieser Art ist die Interaktion zwischen den Entwicklern. Durch `Microservices` ist es möglich auch die Entwicklung selbst in Teams zu unterteilen, die ähnlich wie ihre Dienste über standardisierte Protokolle kommunizieren. Auch erlaubt ein solches System jeden einzelnen Komponenten unabhängig zu skalieren. Wenn man also besonders viele Nutzer hat, die Zugang zu ihren Accounts brauchen kann man einfach den Accounts-Diesten mehr Leistung geben, ohne dass sich für die restlichen Programme und Seiten etwas ändert.
Natürlich ist inzwischen kein einzelner Server in der Lage alle Anfragen an *accounts.google.com* zu verarbeiten. Das wird vor allem ein Problem wenn es um das speichern und erreichen von Zuständen geht. Dieses Thema wird aber noch genauer im Kapitel *distributed systems* angeschaut.

2. Homogene Architektur: Eines der grössten Probleme mit *distributed systems* ist die erhöhte Komplexität. Beispielsweise muss ein Nutzer (Das Frontend) wissen, welchen Service anzufragen ist, um gewisse Daten zu erhalten.
   Wenn man Beispielsweise *accounts.google.com* nach einem `YouTube`-Video fragt, hat das System keine Ahnung was zu tun ist.
   Anders funktionieren Systeme die einer *homogenen Architektur* folgen. In einem späteren Kapitel wird diese Architektur noch sehr genau analysiert, aber momentan sind nur die Grundlagen wichtig:
   Jeder Server (Node) in einem Homogenen System erfüllt die gleichen Funktionen. Man kann also eine beliebige Node anfragen und wird ein Resultat zurück bekommen. Dabei spielt es keine Rolle, ob die Daten auf der richtigen Node sind oder nicht, da das System intern die Daten findet. Diese Architektur ist komplizierter zu implementieren und hat eine vielzahl an Problemen, vor allem wenn es um die Skalierbarkeit geht. Die Details sowie die Programmierung werden wir noch anschauen.
   Natürlich gibt es noch weitere Architekturen, aber für den Moment

reicht es diese beiden Optionen und extremen zu verstehen. Die Alternativen werden wir dann noch im dedizierten Kapitel besprechen.

Teil II

Distributed systems

Auch wenn wir kaum darüber nachdenken oder uns mit den Problemen von *distributed systems* auseinandersetzen, interagieren wir trotzdem täglich unzählige Male mit ihnen.
Tatsächlich lassen sich selbst soziale Interaktionen als eine Art *distributed system* sehen, obwohl in einem solchen System meist die Interaktionen selbst mehr Geschätzt werden als die Änderungen der Zuständen. Als *distributed system* lässt sich eigentlich jeder Zusammenschluss aus mehr oder weniger unabhängigen Prozessen oder Einheiten welche untereinander kommunizieren definieren.
In dieser Arbeit liegt der Fokus allerdings auf *distributed systems* im Zusammenhang mit dem Internet, da diese *relativ* einfach zu entwickeln sind und schnell praktischen Nutzen zeigen. Als erstes ist es dabei wichtig zu verstehen dass schon das Internet selbst bereits ein *distributed system* ist, allerdings geht es hier primär um die Komponenten, die das Internet ausmachen. Denn das Internet ist nicht nur ein Zusammenschluss aus einzelnen Prozessen, sondern aus vielen verschiedenen *distributed systems*.

Die Entwicklung von *distributed systems* läuft schon beinahe so lange wie die Entwicklung von Software selbst. Im Laufe der Zeit gingen die Standards für *distributed systems* durch viele verschiedene Iterationen. Von einfachen mehrstufigen Architekturen bis hin zu modernen, dezentralisierten `peer-to-peer` Netzwerken oder komplett Serverlosen Lösungen.
Im ersten Abschnitt müssen wir uns allerdings die vielen Irrtümer und fehlerhaften Annahmen über *distributed systems* analysieren. Tatsächlich wird der Text leider im Laufe der Zeit nicht wirklich positiver, da es sehr viele Probleme und Nachteile im Zusammenhang mit *distributed systems* gibt. Auch wenn viele verschiedene Varianten und Architekturen für das Entwickeln von *distributed systems* existieren und viele davon für die Situationen in denen sie angewendet werden gut funktionieren, so gibt es trotzdem weder einen *richtigen* noch einigen *perfekten* Weg.
Vor Allem durch das `CAP-Theorem` sehen wir direkt dass es physikalisch unmöglich ist, ein *perfektes* und Nachteil loses *distributed system* zu bauen. Dann stellt sich allerdings wiederum die Frage, wieso solche Systeme so häufig sind, obwohl vieles unmöglich ist und der Rest anscheinend sehr kompliziert zu implementieren ist. Das Internet war Anfangs zumindest ein guter Versuch, da es viele der Entscheidungen den Nutzern überlassen. Auch wenn Menschen natürlich voller Fehler und Nachteile sind, spielen diese keine grosse Rolle solange die Menschen auch die eigentlichen Nutzer des Systems sind. Inzwischen ist dies aber bei weitem nicht mehr der Fall. Datenverkehr durch `Bots` oder `API's` hat den durch Menschen generierten schon lange übertrof-

fen. Auch sehen wir neue Varianten, vor allem für soziale Netzwerke, durch Projekte wie `Mastodon`.
Auch wenn es (zumindest Momentan) physikalisch unmöglich ein *perfektes distributed system* zu Entwickeln, werden wir in Zukunft trotzdem Fortschritte und Innovation in diesem Bereich sehen. Es ist auch zu erwarten dass im Laufe der Zeit schnellere, sicherere und effizientere Netzwerke und Programme sehen werden, aber ohne einen Sprung in Technologien wie beispielsweise Quantum-Computer, kann es wirken, als ob das Feld mehr oder weniger Erschöpft ist.
Es ist wichtig zu verstehen dass wir zumindest Anfangs *Storage* und *Compute* in *distributed systems* genau gleich angehen werden, da jedes *Compute-System* auch *Storage* braucht und jedes *Storage-System* auch Berechnungen durchführen muss. In den dedizierten Kapiteln werden wir dann einige der Unterschiede anschauen und Beispiele und Projekte kennenlernen.

Kapitel 4

Fallacies

Wenn man sich mit *distributed systems* auseinandersetzt wird man einige falsche Annahmen[9] antreffen, die oftmals fälschlicherweise gemacht werden. Viele dieser Annahmen spiegelen sich im `CAP-Theorem` wieder, aber momentan ist es wichtig die einzelnen falschen Annahmen einzeln anzuschauen da, wenn man realisiert dass die Aussagen nie der Wahrheit Entsprechen, es einige wichtige Probleme und Fragestellungen gibt, denen man sich stellen muss, falls man versuchen will sein eigenes *distributed system* zu bauen.
Als gutes Beispiel für ein Missverständnis dieser Gesetze ist der Vortrag von Jimmy Bogard, Avoiding Microservice Megadisasters[20], zu empfehlen.
Auch wenn die genaue Anzahl und Wortlaut der einzelnen Irrtümer umstritten ist, wollen wir hier einige anschauen:

4.1 Das Netzwerk ist ausfallsicher

Am 5. August 1858 wurde das erste Transatlantik-Telegrafenkabel[5] gelegt. Kurz darauf gab es die erste Kontaktaufnaheme zwischen dem Weissenhaus und der englischen Krone. Dieses einfache Kabel verkürzte die Kommunikationszeit von mehreren Wochen auf nur einige Minuten.
Doch bereits zwei Wochen später fiel dieses erste transatlantische Netzwerk aus[6]. Es war unglaublich schwer die Ursachen für das Versagen zu finden und am Ende musste ein neues Kabel gebaut werden, um die Kommunikation fortsetzen zu können.
Auch wenn Netzwerke um ein vielfaches besser wurden zeigt dies doch, dass jedes Netzwerk ausfallen kann und mit genügend Zeit auch ausfallen wird.
Dazu kommmt noch dass man heute kaum noch in Kontrolle über sein kom-

plettes Netzwerk ist. So viele Firmen und Geräte spielen mit wenn es um die Infrastruktur des Internets geht, dass es oftmals überraschend ist, dass überhaupt etwas funktioniert. Die Anzahl der Fehlerquellen ist also sehr hoch und meist reicht ein einzelner Fehler um das gesamte System zum Zerfallen zu bringen.

4.2 Die Latenzzeit ist gleich null

Ebenfalls eines der Irrtümer die im Vortrag von Jimmy Bogard[20] angesprochen werden und welches schnell katastrophale Auswirkungen haben kann. Nehmen wir an, dass unser System aus 50 Dienste besteht. Davon brauchen wir für jeden Request die hälfte aller Dienste. Mit einem Maximum von `50ms` pro Request braucht das Verarbeiten durch unser gesamtes System über eine Sekunde, was meist absolut inakzeptabel ist. Dieses Irrtum ist unteranderem für die Menge an Cacheing Lösungen verantwortlich.

4.3 Der Datendurchsatz ist unbegrenzt

Vor Allem für die ein `fully connected mesh`, welches wir später noch genauer besprechen werden, ist dies ein grosses Problem. Da bei einer solche Architektur die Anzahl der Verbindungen im Quadrat zur Anzahl Nodes steigt und wir dauerhaft `Heartbeat`-Traffic benötigen wird der verfügbare Netzwerkdurchsatz schnell zu einem Problem.

4.4 Das Netzwerk ist informationssicher

Mit immer schnelleren, besseren und einfacheren Cloud-Solutions wird dieser Punkt weniger ein Problem. Trotzdem ist es zu beachten, dass der Verkehr zwischen den einzelnen Nodes genauso, oder noch gefährlicher sein kann, als die Daten die von Aussen kommen. Mit neuen Lösungen wie Google Cloud VPN wird dies einiges einfacher, aber dadurch ist es schwer externe Applikationen ausserhalb von Google in das System zu integrieren. Meist muss man also trotzdem `TLS` oder eine ähnliche Verschlüsselung verwenden um sein Cluster zu sichern.

4.5 Die Netzwerktopologie wird sich nicht ändern

Auch hier muss man akzeptieren, dass dieses Problem kaum zu lösen ist. Neue Nodes werden dazu kommen, wenn mehr Nutzer das System benutzen und andere werden entfernt wenn sie nicht mehr benötigt werden. Auch kann es zu Fehlern oder Abstürzen kommen. Sowohl einzelne Nodes und Gruppen von Nodes als auch das Netzwerk können ausfallen. Daher ist es wichtig, dass es eine gute Infrastruktur für das dynamische Ändern der Topologie gibt. Dafür kann man einfach die Programmiersprache selbst, wie beispielsweise `Erlang` oder externe Tools wie Kubernetes verwenden. Damit zu rechnen, dass die Topologie gleich bleibt könnte verheerende Folgen haben.

4.6 Es gibt immer nur einen Netzwerkadministrator

Für kleinere Projekte ist dieses Irrtum nicht besonders relevant, da es nur manchmal zutrifft. Aber auch hier geht es wieder darum, welchen Datenquellen man vertrauen darf und wer Zugang zu welchen Aktionen hat. Auch hierfür kann man oftmals bereits existierende Lösungen verwenden oder seine eigenen Implementieren, aber es ist klar dass man ohne solche langfristing keine Chance haben wird.

4.7 Die Kosten des Datentransports können mit null angesetzt werden

Tatsächlich kann man hier *Kosten* nicht nur als tatsächliche Kosten gesehen werden. Natürlich kostet der Datentransfer meist etwas. Entweder durch den Internetanbieter oder Cloud Provider kommen eigentlich so gut wie immer Kosten für den Transfer auf. Daher ist es also immer gut, möglich wenig Daten tatsächlich übers Kabel zu schicken. Aber neben den finanziellen Kosten gibt es natürlich auch noch die Kosten, die entstehen, wenn eine Berechnung länger dauert, wodurch andere nicht durchgeführt werden können. Diese Probleme sind schwerer zu finden und beheben und können katastrophale Folgen haben.

4.8 Das Netzwerk ist homogen

Nicht alle Verbindungen in einem Cluster sind gleich stark oder haben eine gleiche Latenz. Beispielsweise könnte ein Cluster über mehrere Kontinente verteilt sein. Dann sind die Verbindungen zwischen den einzelnen Nodes auf dem selben Kontinent natürlich um ein vielfaches schneller als diese, die zuerst über den Atlantik müssen. Wie wir vorhin bereits gesehen haben, sind solche Verbindungen auch weniger stabil und bringen vielerlei andere Probleme, wie beispielsweise höhre Kosten mit sich.
In einer optimalen Welt könnte man jeder seiner Verbindungen in einem Cluster ein *Gewicht* oder *Preis* zuweisen um das verwenden der langsameren Strecken seltener oder schwerer zu machen. Es gibt auch bereits verschiedene Test in diesem Gebiet, aber eine einfache, schnelle Lösung bleibt bislang noch aus.

Kapitel 5

CAP

Über die Jahre wurden immer bessere, schnellere und sicherere *distributed systems* entwickelt. Dabei gab es verschiedene Richtungen in denen geforscht und entwickelt wurde. So gab es *relational databases*, Netzwerke und viele andere Bereiche, die alle ähnliche *distributed systems* bauten, aber ihren Fokus auf verschiedene Bereiche legten.
Natürlich gab es auch solche, die versuchten *„perfekte“*, universale Systeme zu entwerfen. Aber deren Träume wurden im Jahre 1998 zerstört, als `Eric Brewer` seine Arbeit unter dem Namen `CAP principles` veröffentlichte. Auch wenn es heute verschiedene Interpretationen und Auslegungen dieser Prinzipien gibt, gelten sie grundsätzlich trotzdem für alle *distributed systems*.

5.1 Consistency

In einem Cluster bestehend aus vielen verschiedenen Nodes werden Daten meist mehrfach gespeichert sind. Sobald dies der Fall ist, oder das Cluster sich auf eine *nicht-zentrale* Quelle bezieht, muss man sich um *Consistency* kümmern.
Konsistenz in diesem Kontext bedeutet einfach, dass überall immer die gleichen Daten vorhanden sind. Natürlich ist es auch wichtig zu verstehen dass diese Aussage alleine unmöglich umsetzbar ist, da selbst Elektronen Zeit brauchen um von einem Punkt in einem Cluster zu einem anderen zu kommen. Daher muss man noch anfügen, dass entweder die gleichen Daten vorhanden sind, oder bekannt ist dass diese Daten möglicherweise nicht stimmen und daher nicht veröffentlicht werden dürfen.
Mit einem konsistenten System ist es also garantiert, dass man mehrere, beliebige Nodes in einem Cluster nach den gleichen Daten fragt und dann von

allen Nodes auch die gleichen Werte oder einen Fehler zurück bekommt.
Es bedeutet auch dass das Ändern eines Wertes auf einer Node sofort auf alle anderen übertragen werden muss. Am einfachsten geht so etwas wahrscheinlich mit `Locks`. Wenn eine Person also einen Wert ändern will, wird dieser nicht einfach verändert und dann später global überschrieben. Stattdessen werden sowohl der alte, als auch der neue Wert im Speicher gehalten. Dann wird ein Signal an alle betroffenen Server geschickt, mit der Nachricht, dass jemand versucht einen gewissen Wert zu ändern. Alle Server die eine solche Nachricht erhalten schützen dann ihre lokale Kopie des Wertes mit einem `Lock`. Wenn während dieser Zeit dann jemand versucht diesen Wert zu bekommen ist dies nicht möglich da der Wert mit einem `Lock` geschützt ist und der Nutzer bekommt einen Fehler zurück. Sobald alle Server eine solche `Lock` haben und dies dem ursprünglichen Server zurückgemeldet haben, wird der Wert an alle Server geschickt, die ihn dann ersetzten. Dabei bleibt die `Lock` bestehen, da noch nicht garantiert werden kann, dass der neue Wert überall angekommen ist. Das erfolgreiche Ändern des Wertes wird auch wieder dem ursprünglichen Server gemeldet. Sobald dieser von allen betroffenen Servern die Bestätigung erhalten hat, dass der Wert überschrieben wurde, werden die `Locks` wieder nach dem gleichen Prinzip entfernt, da jetzt garantiert werden kann, das überall der gleiche Wert vorhanden ist.
Es wird relativ schnell klar, dass dieser Prozess unglaublich umständlich ist und die Anzahl `Locks` und die Anzahl der benötigten Schritte mit der Anzahl Nodes beziehungsweise der Anzahl Kopien eines Wertes steigt.
Wieso sollte also jemand bereits sein, diesen Aufwand auf sich zu nehmen?
Als gutes Beispiel für *consistency* kann man sich eine Bank vorstellen. Für unser Cluster stellen wir uns Bankautomaten vor, an denen man Geld abheben kann. Dabei hat jeder Bankomat eine Kopie des aktuellen Kontostandes unseres Beispielnutzers. Für unser Beispiel nehmen wir an unser Nutzer hat `$100` auf seinem Konto. Wenn unser Nutzer nun an einem Bankomaten steht und `$75` abhebt soll er auch `$75` bekommen, da er ja genügend Geld auf seinem Konto hat. Sein neuer Kontostand wäre dann also `$25`. Dieser ist aber anfangs nur auf dem einen Bankautomaten vorhanden, auf dem die Abhebung durchgeführt wurde. Wenn dieses Banking-System nicht auf *consistency* gesetzt hätte, wäre es möglich kurz nach der ersten Abhebung eine zweite an einem anderen Automaten zu machen, beispielsweise über `$50`. Dieser hat noch nicht mitbekommen, dass bereits `$75` abgehoben wurden und denkt immer noch, dass der aktuelle Kontostand `$100` sei. Daher würde auch die zweite Transaktion durch gehen und unser Nutzer hätte `$50` gratis bekommen. Natürlich wäre ein solcher Fehler absolut verheerend, wodurch Banken meist auf *consistency* setzen.

5.2 Availability

Manchmal ist es wichtiger, immer halbwegs richtige Daten an den Nutzer zurück zu geben, anstatt die aktuellsten oder richtigen Daten. Ein System welches auf *availability* setzt soll also immer ein Resultat zurück geben. Je nach Definition bezieht sich *availability* auch auf die Antwortzeit. Vor Allem für moderne *Social-Media* Seiten ist diese Eigenschaft eine Priorität. Für viele der Seiten wäre es tödlich auch nur für kurze Zeit nicht verfügbar zu sein.
Tatsächlich ist es relativ einfach festzustellen, dass Seiten wie `Twitter` *availability* über *consistency* gewählt haben. So reicht es einen Tweet auf verschiedenen Geräten zu öffnen und man wird wahrscheinlich eine verschiedene Anzahl Likes und Retweets sehen.
Während sowohl *consistency* als auch *availability* gut mit *partition-tolerance* kombiniert werden können, sind *consistency* und *availability* schwerer zusammen zu bringen. Trotzdem sieht man es in Datenbanksystem wie `Oracle-DB`, auch wenn diese heutzutage seltener sind. Wir werden aber noch genauer besprechen, wie genau diese Eigenschaften kombiniert werden.

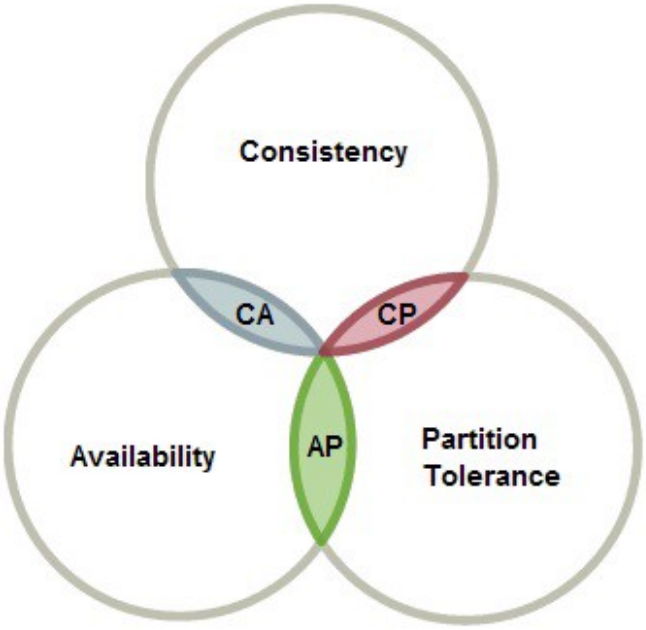
5.3 Partition-tolerance

Ein *distributed system* besteht immer aus verschiedenen Aktoren. Da diese ein zusammenhängendes System bilden ist es zwingend von Nöten das die einzelnen Aktoren Nachrichten zwischen einander senden. Aber was passiert wenn diese Nachrichten nicht ankommen, oder wir nicht garantieren können dass sie ankommen?
Tatsächlich ist es unmöglich zu garantieren, dass eine Nachricht am Ziel ankommt. Man könnte natürlich sagen, dass der Empfänger eine Bestätigung sendet, sobald er die Nachricht erhält, aber dann haben wir das Problem, dass wir nicht garantieren können dass diese Bestätigung ankommt. Dieses Problem geht dann weiter und ist unlösbar. Es ist auch unter dem Namen *The Two Generals' Problem* [24] bekannt. *Tom Scott* hat eine der besten Erklärungen, und hat das Problem auch noch gut visualiziert.
Neben dem einfachen Fakt dass wir nicht garantieren können, dass unsere Nachrichten ankommen, gibt es natürlich haufenweise Möglichkeiten wie der Sender oder Empfänger abstürzen können. Dazu kommt noch das Netzwerk zwischen den beiden das ebenfalls von Problemen betroffen ist.
Wenn ein System auf *partition-tolerance* setzt ist es also möglich, dass eine

beliebige Anzahl Nachrichten nicht ihr Ziel erreichen, ohne dass das System als ein ganzes betroffen wird.
Partition-tolerance betrifft auch das Problem eines Network-Splits. Also was passieren soll, wenn sich unser Cluster in zwei Teilt.
Auch wenn sich Netzwerktechnologien laufend verbessern muss man trotzdem davon ausgehen, dass jedes Cluster und jede Lösung *partition-tolerance* in irgend einer Form unterstützen muss. Natürlich gibt es auch Plattformen, die nicht auf *partition-tolerance* setzen, diese fallen dann aber meist schnell auseinander, sobald Probleme mit dem Netzwerk aufkommen oder sind nur auf einer Node verfügbar.

5.4 Varianten

In einfachster Form kann man sagen, dass man aus diesen drei Eigenschaften zwei auswählen soll. Auch wenn es kleine Tricks und optimierungen gibt, sagt `Eric Brewer's` Arbeit auch aus, dass es unmöglich ist, alle drei Eigenschaften in einem System zu haben. Man kann sich die drei Eigenschaften auch grafisch vorstellen:



Wie bereits angesprochen ist es unmöglich alle drei Eigenschaften gleichzeitig

durchzusetzen. Auch ist sind Probleme mit Netzwerkprobleme und Fehler beim senden von Nachrichten beinahe unmöglich zu verhindern. Daher hat man meistens die Wahl zwischen `CP` (*Consistency-partition-tolerance*) und `AP` (*availability-partition-tolerance*).

1. CP: Wir haben im Abschnitt zu *consistency* haben wir bereits etwas über Banken-Systeme gesprochen. Wie bereits ein Beispiel mit den Bankomaten angesprochen könnten die Folgen verheerend sein sollte ein Bank-System nicht auf *Consistency* setzten.. Tatsächlich sieht man diese Option nicht nur bei Banken, sondern beispielsweise auch bei Online-Stores oder Ticket Verkäufen. Da es theoretisch möglich ist, dass ein System ohne *consistency* ein Ticket oder Produkt mehrfach verkauft ist dies natürlich keine Option für solche Seiten.

2. AP: `AP` ist die häufigste Wahl für moderne Webseiten. Es geht vielen dieser Seiten mehr darum, so vielen Nutzern wie möglich ungefähr richtige Resultate Resultate zu liefern. Als gutes Beispiel für die `AP` Kategorie gilt auch das `DNS`-Netzwerk. Es kann oftmals mehrere Tage dauern, bis neue `DNS-Records` durch die gesamte Hirarchie geändert werden. Aber sollte `DNS` global auch nur für wenige Sekunden nicht verfügbar sein wären die Auswirkungen unvorstellbar schlimm.
   Und trotzdem kann ein System, welches nicht auf *consistency* setzt in einen Zustand geraten in dem Langfristig verschiedene Wahrheiten vorhanden sind. Man muss also auch ein System einbauen, welches sich darum kümmert diese Probleme zu beheben und welches Entscheiden kann wie man verschiedene Daten zusammenführt oder welche Variante gewählt wird.

3. Mischungen: Tatsächlich gilt das `CAP-Theorem` nur für einen einzelnen Moment. Da es längerfristig trotzdem wichtig ist, die Werte richtig zu speichern und alle Events mit zu zählen wählen die meisten Systeme eine Mischung aus `CP` und `AP`. In einer Datenbank wird also beispielsweise direkt nach dem schreiben eines Wertes etwas *inconsistency* akzeptiert. Dafür laufen langfristig Prozesse um alle Werte wieder gleich zu bekommen. Als Beispiel dafür hat `Riak Anti-Entropy`, welches dauerhaft im Hintergrund läuft.

Kapitel 6

Architekturen

In den vorherigen Abschnitten haben wir die Probleme und Eigenschaften von *distributed systems* angeschaut, sowie einige Applikationen und Beispiele kennen gelernt. Tatsächlich gibt es tausende von verschiedenen Implementierungen und Varianten. Sowohl durch die verschiedenen Eigenschaften des `CAP-Theorems`, als auch durch das setzen von Prioritäten wenn es um die Irrtümer und Funktionen geht, gibt es eine vielzahl an verschiedenen Varianten. Dazu kommt noch, dass auch bei gleichen Eigenschaften noch verschiedene Fokuspunkte gesetzt werden können.
Im laufe der Jahre entstanden viele verschiedene Architekturen für das Entwickeln von *distributed systems*. Dabei sind manche, wie `Microservices` der dominante Standard, während andere wie `Blockchain` momentan noch rein experimentell sind.

6.1 Monolith

Um in nächsten Abschnitt `Microservices` richtig erklären zu können muss als erstes die *Vorgänger-Architektur* besprochen werden. Auch wenn heute noch `Monolithen` gebaut werden, haben alle grossen Firmen und Projekte inzwischen auf `Microservices` oder eine andere Architektur gewechselt.
Tatsächlich passen `Monoliths` nicht wirklich in diese Liste, da sich der Begriff eigentlich mehr auf die Codebase anstatt die Infrastruktur bezieht. Eigentlich verwendet man den Begriff im Zusammenhang mit einer einzelnen, zentralen Codebase. Aber tatsächlich ist die Struktur der Codebase irrelevant für die Architektur und die gewählte Infrastruktur. Mit einem `Monolith` beschreibt man meistens eine einzelne Codebase, die das gesamte Projekt ausmachen.

Aber im Laufe der Zeit wurde Monolith zu einem Synonym für ein System, welches auf einer einzigen Maschine läuft, meist mit altmodischen Sprachen und Tools.
Natürlich hat dies viele Vorteile. Nur eine Sprache, nur ein Build-tool, nur eine Maschine zu verwalten und nur ein Punkt um nach Problemen zu suchen. Aber mit mehreren Entwicklern und komplizierten Projekten kann dies natürlich problematisch werden. Dazu kommt natürlich auch, dass viele Programme nicht nur von einer Maschine unterstützt werden können.

6.2 Microservices

Während ein Monolith für das komplette Projekt verantwortlich ist, so übernimmt ein Microservice nur eine einzelne Aufgabe. Tatsächlich sind die Unterschiede zwischen den beiden überraschend gering. Ein Microservice folgt den gleichen Kriterien und Eigenschaften wie auch ein Monolith, aber er ist Teil eines grösseren Systems.
Für jeden Service gelten also die gleichen Kriterien wie oben bereits angesprochen, nur auf einer anderen Grössenordnung.
Der Vorteil eines eines solchen Systems ist das jeder Komponent unabhängig vom restlichen Systems operiert. Da zwischen allen ein universeller Standard zur Kommunikation verwendet wird, meist wählt man HTTP und JSON, spielt es keine Rolle ob verschiedene Microservices die gleiche Programmiersprache verwenden.
Diese Unabhängigkeit ist natürlich auch während der Entwicklung hilfreich. So erlauben Microservices das Entwickeln eines Projektes mit verschiedeneen Teams und theoretisch auch Firmen. Da die einzelnen Dienste getrennt laufen ist es auch möglich, einen dieser Dienste in mehreren Projekten zu verwenden. Als Beispiel dafür haben wir bereits /accounts.google.com/n angeschaut.
Aber für viele können Microservices nur wie einfachere Monolithen wirken. Dabei vergisst man gerne die Irrtümer. So besagt eines der Irrtümer, dass die Latenzzeit gleich null ist. Mit verschiedenen Diensten die alle über relativ langsame Protokolle wie HTTP kommunizieren müssen kann diese falsche Annahme schnell verheerende Auswirkungen haben. Als gutes Beispiel für diese Art von Fehlern lässt sich der Vortrag von Jimmy Bogard, Avoiding Microservice Megadisasters[20], empfehlen. Darin erklärt er was passieren kann, wenn man die Irrtümer nicht als Irrtümer, sondern als Fakten oder Regeln nimmt.
Tatsächlich passen auch Microservices nicht ganz in diese Liste, da der

Begriff lediglich von einer Gruppe von unabhängigen Diensten redet. Wie genau man diese Dienste dann baut ist nicht geregelt. Aber natürlich kann man sagen, dass diese Gruppierung von Diensten ein *distributed system* bildet.
Über Jahre entstanden dann trotzdem Richtlinien und Regeln wie man solche `Microservices` zu bauen hat. So ist es inzwischen der akzeptierte Standard `Microservices` *stateless* zu bauen. Für weniger fortgeschrittene Programmiersprachen und virtual machines ist dies der einfachste und schnellste Weg einen fertigen Service zu bauen. Man geht als Grundlage einfach davon aus, dass jede Instanz eines `Microservice` ohne irgendwelche Daten arbeitet. Tatsächlich ergibt diese Annahme auf den ersten Blick sogar etwas Sinn. Da auch `HTTP` ein *stateless* Protokoll ist, lohnt es sich seine Dienste ebenfalls so zu bauen. Jeder Request an einen solchen `Microservice` muss also alle Daten enthalten die es braucht um den Request zu erfüllen. Meist wird dafür einfach eine `ID` gesendet, mit der dann die restlichen Daten von einer Datenbank oder anderen Dinest geholt werden können. Der Vorteil dieses Systems ist es, dass die einzelnen Instanzen unabhängig voneinander operieren. Daher kann man sie einzeln starten und stoppen oder neue hinzufügen. Vor Allem für das skalieren ist diese Eigenschaft besonders hilfreich.
Aber sobald man anstatt `HTTP` ein Protokoll verwendet, welches nicht *stateless* ist, zerfällt das gesamte System. Dann muss man etwas unbeholfene Notfall lösungen wie *Redis* oder etwas ähnliches verwenden. Als gutes Beispiel für ein solches Protokoll kann man sich `WebSockets` anschauen. Diese erlauben Echtzeit Kommunikation zwischen Client und Server. In Zukunft werden immer mehr Seiten in Echtzeit laufen und es wird immer mehr Verwendung für `WebSockets` geben. Da diese aber nicht gut zu `Microservices` passen kann man davon ausgehen, dass Alternativen gefunden werden müssen, oder es eine Grundsätzliche änderung in der Art der Entwicklung von `Microsersvices` geben wird, beispielsweise durch die Verwendung von `Erlang` und `Elixir`, die wir uns später noch anschauen werden.

6.3 Client-server

`Client-server` ist das wahrscheinlich einfachste Modell, das ebenfalls nur bedingt gut in diese Liste passt. Das Internet selbst ist ein *distributed system* dass das `Client-server` Modell nutzt. Dabei sind die Clients meist Browser und die Server sind Webserver. Die meisten kleinen `Monolithen` verwenden dieses System, bei dem es viele Clients aber nur einen Server gibt. Auch sagt das Modell nichts über die Architektur innerhalb des Servers aus

und beschreibt keinerlei Mechanismen zum lesen, finden oder schreiben von Daten. Die meisten anderen Modelle in dieser Liste fokussieren sich rein auf die Server Seite. Trotzdem wird das Modell in leicht umformulierter Form verwendet. Als gutes Beispiel dafür werden wir uns später noch `Apache Kafka` anschauen, welches ebenfalls Clients und Server hat. Der Server ist dabei ein Consumer, während der Client `Kafka` selbst ist.

6.4 Three-tier

Eine weitere häufige Architektur ist das so genannte `Three-tier` Modell. Eigentlich gibt es auch noch das `n-tier` Modell, aber das grundlegende Konzept ist bei beiden das gleiche. Auch dieses Modell sagt nichts über die Architektur innerhalb der eigentlichen Server (oder hier: *tiers*) aus, sondern legt eher fest, wie man mehrere *tiers* anordnen soll. Aber da die meisten `Microservice` sowie viele andere Seiten und Applikationen dieses System verwenden ist es trotzdem wichtig einen Blick darauf zu werfen:

- Presentation tier: Im Fall von Web Development ist das presentation tier meist ein Server für statische Dateien wie beispielsweise `Apache` oder `Nginx`. Dieses *tier* ist einfach dafür verantwortlich das Interface an den Nutzer zu liefern und die Inputs in ein Format um zu wandenln, dass das *logic tier* verstehen kann (Es ist auch wichtig zu verstehen, dass jede Anfrage und jegliche Daten nur von einem *tier* zum nächsten gehen können und man niemals vom *presentation tier* direkt auf das *data tier* zugreifen sollte).

- Logic tier: Dieses *tier* ist meist die eigentliche Applikation. Hier werden Anfragen verarbeitet und Authentication sowie Authorization überprüft. Oftmals wird dieses *tier* alleine *Backend* genannt.

- Data tier: Meist denkt man nicht besonders gross über dieses *tier* nach, da oftmals eine einfache Datenbank genügt. Aber das dritte *tier* befasst sich mit allen Aufgaben zum Speichern von Daten. Wir haben bereits gesehen, dass Daten aber nicht immer in einer klassischen Datenbank gespeichert werden müssen, wodurch die differenzierung zwischen *tier 2* und *tier 3* schwammig werden kann.

Die obige Erklärung befasste sich nur mit normalen Webservern, wie `Microserivces`. Aber im Abschnitt zu `Meta-Clustering` wollen wir uns das `Whatsapp` Modell genauer anschauen. Diese Teilen ihren `three-tier` etwas anders auf:

- Presentation tier: Damit wird ist Falle von `Whatsapp` keine `HTML` Datei gemeint, sondern der Server der die eingehenden `TCP`-Verbindungen Verwaltet. Dieser sorgt eigentlich nur für das Senden und Empfangen der einzelnen Nachrichten.

- Logic tier: `Whatsapp` verwendet für dieses *tier* mehrere Server, aber einfach gesagt ist es der Zusammenschluss aus allen Clustern und Servern die sich mit der Logik befassen, mit denen Nutzer in der App interagieren.

- Data tier: `Whatsapp` verwendet für das Speichern von Chatnachrichten `Mnesia` (Eine Datenbank von Erlang-Solutions die wir noch genauer anschauen werden). Dadurch speichern sie ihre Daten auf den Servern auf denen auch das *logic tier* operiert. Trotzdem ist es wichtig zumindest während der Entwicklung eine klare Separation zwischen den beiden Ebenen zu haben, da sonst die einzelnen *tiers* schnell vermischt werden können.

6.5 Fully connected mesh

Alle Systeme die wir bisher angeschaut haben funktionieren meist *stateless*. Das bedeutet dass sich keinerlei Daten im laufenden Programm selbst befinden, sondern für jede Aktion alle direkt aus der Datenbank oder von den Nutzern geholt werden. Damit löst man eigentlich überhaupt kein Problem. Man verschiebt es lediglich auf die Datenbank, da man davon ausgeht dass diese besser damit zurecht kommt. Aber es gibt viele Situationen in denen das einfach keine Option ist:

- Es ist möglich Programme und Webserver komplett ohne externe Datenbank zu bauen. Der primäre Vorteil davon ist natürlich dass man kürzere Antwortzeiten hat, da keine externe Quelle angefragt werden muss. Je nach Netzwerk und Server kann eine *Query* auf einer Datenbank teure Millisekunden kosten, die meist direkt auf den Nutzer übertragen werden, wodurch sich die Erfahrung beim verwenden der Seite verschlechtert. Auch stellt die Datenbank einen Fehlerpunkt dar, den man vermeiden sollte. Die meisten Webserver und Seiten fallen komplett aus sobald Probleme mit ihrer Datenbank entstehen.

- Manche Daten werden so häufig verwendet und geändert, dass man sie nicht effizient in eine Datenbank schreiben kann. Diesen Kritikpunkt

kann man umgehen in dem man einen Cache wie `memcached` verwendet, aber damit löst man auch das Problem nicht, man minimiert nur die Auswirkungen. Als gutes Beispiel für dieses Problem kann man sich `WebSockets` oder `TCP-Sockets` vorstellen und man muss die `ID's` der Verbindungen speichern. Da sich diese Dauerhaft ändern und dauerhaft verwendet werden ist es nicht Ratsam sie in einer klassischen Datenbank zu speichern.

- Auch ist es einfach schneller ein System zu entwickeln, wenn man sich nicht um die Datenbankverwaltung kümmern muss. Viele der älteren, grossen Technologieunternehmen haben in jedem Team noch einen Datenbank-Administrator. Wenn aber die Methode zur Speicherung direkt in der Applikation selbst integriert ist fällt eine externe Abhängigkeit komplett weg.

Momentan reden wir nur vom Speichern und Finden von Daten, da man auch das Verarbeiten und Berechnen von Werten so ausdrücken kann (Ein Prozess der eine Berechnung durchführt hat immer auch eine `ID` die gespeichert und gelesen werden muss).
Um keinen zentralen Fehlerpunkt haben sollen die Daten also über das gesamte Cluster verteilt gespeichert werden. Dabei gehen wir davon aus, dass jeder Wert nur ein einziges mal gespeichert wird. Wie man das gesamte Cluster tatsächlich Fehlertolerant machen kann werden wir im Kapitel zu *Storage* noch genauer anschauen.
Da aber unsere Daten zufällig über das gesamte Cluster verteilt sind müssen wir ein System haben diese zu finden. Theoretisch ist es möglich, gewisse Nodes im Cluster mit zufälligen Anderen zu verbinden und wir werden das im nächsten Abschnitt noch anschauen.
Um es also möglich zu machen, Daten einfach von allen Nodes im Cluster erreichbar zu machen werden wir ein `fully connected mesh` verwenden. Wie der Name bereits sagt, sind in einem solchen Cluster alle Nodes mit allen anderen verbunden. Daher reicht für jede Nachfrage nur ein Sprung um beim Ziel zu sein. Natürlich ist dieses System optimal, da man jeden Wert in kürzester Zeit erreichen kann. Allerdings hat es auch einige Nachteile:

- Heartbeat: Da jede Node mit jeder anderen Verbunden ist, muss jede Node dauerhaft testen ob eine der verbunden Nodes abgestürzt ist oder noch verfügbar ist. Da es momentan kein System für *konstante* Verbindungen gibt, ist es momentan der Standard *Heartbeating* zu verwenden. Dafür werden in regelmässigen Abständen Signale an alle Nodes geschickt, die diese dann bestätigen. Dadurch lassen sich Ausfäl-

le schnell erkennen. Dieses System ist, so wie viele andere auch, nicht tatsächlich *real-time*. Es benötigt immer noch die einzelnen *heartbeats*. Wenn also zwischen zwei *beats* etwas geschieht, dauert es bis zum nächsten um den Fehler zu erkennen. Aber umso geringer die Frequenz der *heartbeats*, desto mehr Daten müssen über das Netzwerk gesendet werden.

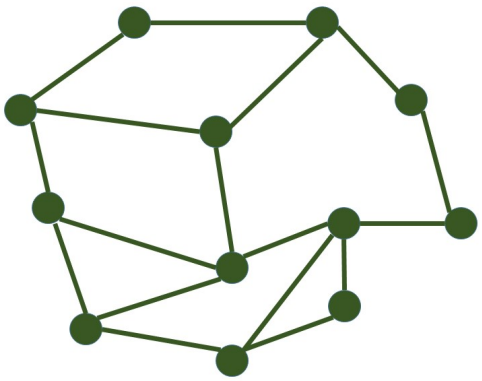
- n^2: Für die Erklärung von diesem Problem gehen wir der Einfachheit halber davon aus, dass tatsächlich *konstante* Verbindungen zwischen den Nodes existieren. Ob dies in der Realität tatsächlich umsetzbar ist oder nicht spielt momentan keine Rolle.
  Da jede Node mit jeder anderen verbunden ist, steigt die Anzahl der Verbindungen n^2 mit n Nodes. Für kleinere Cluster ist dies kein Problem. Mit drei Nodes sind neun Verbindungen auch für die schlechtesten Netzwerke machbar. Aber grosse Seiten wie `Facebook` haben geschätzt zwischen 80000 und 160000 Nodes. Für ein solches Cluster müsste ein Netzwerk also 25600000000 Verbindungen unterstützen. Selbst die besten und stärksten Netzwerke haben keine Chance diese Anzahl Verbindungen zu unterstützten. Dazu kommt auch dass die Nodes selbst ebenfalls eine grosse Anzahl Verbindungen unterstützen müssen. Zwar nicht n^2, aber trotzdem n.
  In einem nächsten Abschnitt werden wir uns noch `Riak` und `Riak-Core` anschauen. Für die Entwicklung dieser wird eine maximale Clustergrösse von etwa 65 angenommen. Für `Erlang`, welches wir später auch noch kennen lernen werden gab es bereits Experimente mit bis zu 500, aber für die meisten Systeme wird nicht mehr als 100 empfohlen.

Für die meisten Projekte sind die obigen Nachteile nicht relevant. Mit `Erlang` und `Elixir` ist es gut möglich bis zu zwei Millionen Nutzer auf einem einzigen starken Server zu unterstützen. Ein gutes Cluster sollte also mindestens 100 Millionen schaffen. Trotzdem gibt es immer wieder Seiten die darüber hinaus wachsen, oder intensivere Aufgaben erledigen müssen, wodurch die maximale Anzahl Nutzer natürlich schnell sinkt.

6.6 Peer-to-peer

In diesem Abschnitt wollen wir einen genaueren Blick auf die *Server peer-to-peer* Architektur werfen. Anders als die, im Abschnitt *Blue sky* besprochene, *Client peer-to-peer* Architektur geht es also wiederum um einen Zusam-

menschluss von Servern, also ein Cluster.
Im Abschnitt zu `fully connected mesh` haben wir bereits das n^2 Problem kennen gelernt. Mit einer `peer-to-peer` Strategie versucht man meist, dieses Problem zu umgehen und die maximale Anzahl Verbindungen und Server pro Cluster zu erhöhen.
Man kann sich diese Architektur einfach gesagt als `fully connected mesh` vorstellen, bei dem einige Verbindungen ausgelassen wurden. Also anstatt des klassischen Kreises mit dem ein `fully connected mesh` gerne visualisiert wird, wird diese Architektur meist anders dargestellt: [8]



Mit dieser neuen Architektur kommen aber einige neue Probleme auf:

1. Traversal: Während wir mit einem `fully connected mesh` garantieren können, dass jeder Wert mit nur einem Netzwerk Sprung erreicht werden kann, ist dies bei einem `peer-to-peer` Layout nicht der Fall. Wir brauchen also eine Strategie und ein Programm, dass es uns erlaubt Daten in einem, nicht regelmässigen, Netzwerk zu finden.

    - Am einfachsten ist natürlich, dass ganze einfach Zufällig zu machen. Dafür sendet man von der ersten Node ein Signal an alle verbundenen, mit der Frage nach der gesuchten Ressource. Wenn diese die Anfrage nicht erfüllen können machen sie einfach das gleiche und senden die Anfrage ebenfalls weiter. Wenn die Anfrage bei einer Node dann Erfolgreich ist, wird der gleiche Weg zurück genommen, bis am Ende die Ressource an der ersten Node angekommen ist.

Meist nimmt man auch nicht alle Verbindungen, sondern nur eine gewisse Anzahl, da sonst das Netzwerk schnell überlastet werden kann. Man muss also eine Wahl zwischen Geschwindigkeit und Anzahl Requests machen.
Zusätzlich muss man, um diese Lösung funktionsfähig zu machen, eine Art *Counter* oder *Log* in die Anfrage einbauen, da es sonst passieren könnte, dass eine Anfrage im Kreis herum gegeben wird. Es werden also eine vielzahl von Mechanismen und Entscheidungen benötigt, um diese Lösung überhaupt zum Laufen zu bringen.

- Je nach Layout der Daten ist es auch möglich, die Verbindungen selbst logisch zu Ordnen. Beispielsweise könnte jede Node einen gewissen Zahlenbereich an Schlüsseln abdecken (Wie man so etwas berechnen kann wird später noch angesprochen). Dann wüsste jede Node, an welche Verbindung ein Request weitergeleitet werden muss, um das richtige Ziel zu erreichen. Diese Variante funktioniert natürlich nur dann, wenn alle Werte einen gleichmässig verteilten Schlüssel besitzen. Auch ist es nicht möglich die Daten zu durchsuchen oder Queries auszuführen, ohne jede Node im Cluster zu befragen.
- Theoretisch ist es auch möglich, eine zentrale Instanz zu haben, welche die Indexe aller Daten kennt. Diese könnte dann optimale Routen berechnen und Anfragen leiten. Die Problematik mit dem Anlauf wird im Abschnitt `Layout` noch genauer besprochen.

2. Network split: Da je nach Programm und Infrastruktur die Bildung der Verbindungen zufällig ist, kann es geschehen, dass sich das Cluster in zwei Gruppen teilt. Beispielsweise könnte die Verbindung zweier Gruppen an nur einer, beziehungsweise zwei Nodes liegen. Sollte eine der beiden ausfallen entsteht ein `network split` (Zwar ähnlich aber nicht gleich zu einem `network split` durch ein Fehler im Netzwerk, da diese meist nicht die interne Architektur repräsentieren. Dabei existieren zwei unabhängige Gruppen des Clusters die nicht miteinander Kommunizieren können und meist nicht einmal von der Existenz der anderen Gruppe wissen. Daher ist es auch sehr schwer zwei getrennte Gruppen wieder zu verbinden, wodurch der Prozess meist manuell durchgeführt werden muss. Im obigen Abschnitt war bereits kurz die Rede von einer zentralen Instanz für das Verwalten des Clusters. Die existenz eines solchen *Management-Servers* würde dieses Problem natürlich lösen, aber wie wir gleich sehen werden hat auch dieser Anlauf einige Probleme.

3. Layout: Es ist immer am einfachsten, das Problem zu verschieben und temporär zu lösen, anstatt eine tatsächliche, anhaltende Lösung zu finden. Dabei übergeben wir das Problem meist anderen, die sich darum kümmern müssen. Nur realisieren wir oftmals nicht, dass diese *anderen* nur wir selbst in der Zukunft sind.

   Ein solcher *temporärer* Fix ist auch das verwenden eines zentralen *Management-Servers*, der das gewünschte `Layout` des Clusters kennt und die Verbindungen zwischen den Servern verwalten und überwachen kann.
   Auf den ersten Blick wird diese Lösung nur positiv und nimmt viel Druck von den einzelnen Servern die sich um viel weniger kümmern müssen. Aber tatsächlich löst ein *Management-Server* überhaupt keine Probleme langfristig:

   - SPOF: Mit einer *zentralen* Instanz hat man genau das. Einen *zentralen* Punkt von dem das restliche System abhängig ist. Dadurch entsteht ein SPOF (*Single point of failure*). Aber tatsächlich wirkt auch dieses Problem lösbar. Dieser gesamte Text befasst sich mehr oder weniger damit, wie man aus mehreren Servern einen einzigen homogenen Körper macht. Aber wenn wir aus unserem *Management-Server* ein Cluster machen, haben wir eine gewisse rekursive Logik.
   - Rekursion: Nun haben wir plötzlich ein Cluster als *Management-Server* (Oder inzwischen: *Management-Cluster*). Anfangs können wir dafür einfach ein klassisches `fully connected mesh` verwenden. Aber wir haben bereits gesehen, dass Facebook bis zu 160000 Server zu verwalten hat. Das ist wohl kaum mit nur 65 Servern zu machen. Also brauchen wir für unser *Management-Cluster* eine andere Architektur. Dafür könnten wir wiederum `peer-to-peer` wählen. Um das ganze effizient zu machen installieren wir wiederum einen *Management-Server*.
     Es wird schnell klar, dass dies keine langfristige Lösung ist und das Problem lediglich vor sich her schiebt. Dazu kommt noch, dass mit jeder weiteren Node und Ebene geringe ineffizienzen im System stärker sichtbar werden.

4. Discovery: Im oberen Abschnitt haben wir bereits einige ziemlich fundamentale Fehler und Probleme in `peer-to-peer` Netzwerken festgestellt. Trotzdem sind diese überraschend häufig. Das Internet selbst

kann als ein solches gesehen werden. Aber das Internet ist für Menschen ausgelegt. Die *Service-Discovery* funktioniert also über Menschen, die sich `Domänen` auswendig lernen, sie von Freunden hören oder Suchmaschinen wie `Google` verwenden (Innerhalb von `Google` existieren natürlich die gleichen Probleme und man kann selbst die Suchmaschine als *zentralen* Fehlerpunkt sehen. Sollten wir versuchen ein *optimales distributed system* zu bauen wäre dies also keine Option. Häufig wird dafür ein dedizierter Dienst verwendet. Jemand der also einen Dienst hat, der von anderen verwendet werden soll kann diesen also in die Liste im dafür verantwortlichen Dienst schreiben. Jemand der dann einen solchen braucht kann einfach in der Liste einen aussuchen. Aber *dedizierter Dienst* ist lediglich ein schönerer Begriff für zentrale Instanz. Und die Probleme, die durch das verwenden von solchen kommt, haben wir bereits angesprochen.
Als alternative wird meist `Broadcasting` eingesetzt. Sobald also ein neuer Dienst verfügbar ist, wird eine Info an alle verbundenen Nodes gesendet (Wodurch wieder Probleme mit dem traversal aufkommen). Diese können dann entscheiden was sie mit der neuen Information anfangen wollen. Aber auch dafür muss man Ausnahmen beachten. Beispielsweise könnte es passieren, dass eine Node während einer solchen Ankündigung temporär offline war, oder erst danach online kam. Wie man mit solchen Problemen umgeht wird im Abschnitt zum *Joining* von Nodes besprochen.

5. Joining: Gehen wir davon aus, dass wir ein Cluster ohne *Management-Server* haben, da wir ja bereits einige Probleme mit solchen festgestellt haben. Wie fügen wir aber nun eine neue Node zum Cluster hinzu?

    - Zufall: Wenn wir eine Art *Service-Discovery* für die einzelnen Nodesae haben, können wir einfach einige zufällig auswählen, uns mit diesen verbinden und hoffen dass sich das Netzwerk mehr oder weniger gleichmässig ausbaut. Aber oftmals ist ein solcher Mechanismus nicht vorhanden, oder wir wollen einen so wichtigen Aspekt des Clusters nicht dem Zufall überlassen.
    - Kopieren: Als Alternative ist es auch möglich, eine komplette Node zu kopieren. Dabei würde die neue Node lediglich die Verbindungen übernehmen, nicht die tatsächlichen Daten. Theoretisch sollte es dadurch möglich sein, anfangs von Hand gleichmässige Verbindungen aufzusetzen und danach einfach die existieren-

den zu kopieren. Natürlich kann dies dazu führen, dass neue Verbindungen nur zu älteren Nodes aufgebaut werden, wodurch das Netzwerk wiederum ungleichmässig wird.

- Expansion: Diese Variante benötigt wiederum eine Art *Service-Discovery* für Nodes um korrekt zu funktionieren. Anders als die vorherigen beiden Startet eine Node diesmal komplett ohne Verbindungen oder mit sehr wenigen, zufälligen. Trotzdem soll sie in der Lage sein externe Anfragen zu verarbeiten. Sollte nun also jemand einen Wert von der neuen Node verlangen, den die Node nicht finden kann wird eine neue Verbindung aufgebaut zu der Node die in der Lage ist die Informationen zu liefern. Man muss dann lediglich ein maximum an Verbindungen festlegen und das Netzwerk wird sich im Laufe der Zeit selbst ausbauen. Aber auch mit diesem Anlauf kommen zwei offensichtliche Probleme auf:
  - Der oben genannte Schritt „... wird eine neue Verbindung aufgebaut zu der Node die in der Lage ist die Informationen zu liefern“ klingt äusserst einfach. Aber es stellt sich die Frage, wie die Node bestimmt wird, die tatsächlich die richtigen Daten besitzt. Tatsächlich kann man zum lösen dieses Problems einfach wieder auf den Beginn des Kapitels `peer-to-peer` verweisen, da es das gleiche Problem darstellt. Daher trifft man auch dabei wieder auf die gleichen ineffizienzen oder Probleme mit zentralen Instanzen.
  - Da die neue Node in der Lage ist externe Anfragen zu verarbeiten ist es theoretisch möglich, dass neue Daten auf dieser neuen Node gespeichert werden, bevor irgendwelche Verbindungen zu anderen Nodes gemacht wurden. Sollte dies geschehen sind die Daten für den Rest des Clusters komplett unerreichbar, bevor die Node nicht einige Verbindungen herstellt.

6.7 Meta-Clustering

Der Begriff `Meta-Clustering` wurde ursprünglich durch `Whatsapp` bekannt. Der Chat-Service hat inzwischen über zwei Milliarden Nutzer und ist mehr ausgelastet als das globale `SMS`-Netz mit mehr als 30 Millionen Nachrichten pro Minute. Tatsächlich verwendet `Whatsapp` für alle Chat-Dienste `Erlang`. Wir werden die Sprache später noch genauer kennen lernen, aber momentan reicht es zu verstehen, dass auch `Whatsapp` auf ein

fully connected mesh setzt. Da jeder Nutzer jedem anderen zu jedem Zeitpunkt eine Nachricht schicken könnte und man inzwischen erwartet möglichst kurze Latenzzeiten zu haben war dies die richtige Entscheidung. Aber mit so vielen Nutzern ist es beinahe unmöglich alle mit nur einem einzigen Cluster zu unterstützen. Um dieses Problem zu lösen kamen die Entwickler mit dem Konzept von Meta-Clustering auf. Auf den ersten Blick ist Meta-Clustering nur ein besonderer Begriff für Microservices. Und es gibt tatsächlich viele Ähnlichkeiten:

- Das gesamte System wird in verschiedene Komponenten aufgebrochen die sich je um eine Aufgabe kümmern.
- Jeder Komponent läuft mehr oder weniger unabhängig von den anderen.
- Die einzelnen Komponenten kommunizieren über ein standardisiertes Protokoll.

Aber anders als die meisten Microservices verwendet Whatsapp ein low-level TCP Protokoll für den Datenaustausch. Dadurch ist es einfacher und schneller Daten zwischen den Clustern aus zu tauschen.
Beispielsweise ist die Aufgabe eines der Cluster nur die Websocket Verbindungen zu den Geräten aufrecht zu erhalten und jeder Verbindung eine ID zu geben. Dann gibt es einen weiteren Dienst der nur dafür verantwortlich ist, die ID's mit den dazugehörigen Telefonnummern zu speichern. Ein weiterer kümmert sich um Push-Notifications, ein anderer um Gruppen und so weiter. Da die einzelnen Cluster stärker verbunden sind verliert man natürlich einiges an Unabhängigkeit. Für Whatsapp ist dies aber kein so grosses Problem, da die Cluster an sich aus vielen Erlang Nodes bestehen und dadurch eher selten komplett ausfallen.

6.8 Blockchain

Viele der hier besprochenen Konzepte behandeln *dezentralisierte* Applikationen. Es ist inzwischen wahrscheinlich nicht mehr möglich, den Begriff *dezentral* in den Mund zu nehmen, ohne nicht Bitcoin und Blockchain zu erwähnen.
Aber es lohnt sich nicht zu viel Zeit mit dem Thema zu verschwenden, da die Technologie nicht wirklich auf unsere Aufgaben passt.
Das Bitcoin Netzwerk braucht über eine Stunde und bis zu einem Tag um eine einfache Transaktion durchzuführen.

Bitcoin liefert absolute *dezentralisierung* und Sicherheit, aber die geringe Geschwindigkeit und Kapazität macht es so gut wie unbrauchbar für *distributed systems* wie wir sie hier besprochen haben.

6.9 Blue sky

Eine weitere sehr interessante Architektur für *distributed systems* ist die Blue sky Architektur. Der Name an sich ist bereits äusserst interessant: Inzwischen laufen beinahe alle grossen Seiten und Dienste auf der Cloud. Die Cloud ist dabei nichts anderes als ein Server, der von einem Cloud-Provider betrieben wird (Die grössten Cloud-Provider: AWS, GCP und Azure). Es ist einiges günstiger Server von diesen Firmen zu mieten, statt eigene zu kaufen oder gar zu bauen. Aber inzwischen laufen (je nach Schätzung) zwischen 3% und 50% des gesamten Internets auf AWS alleine. Sollte es also einen katastrophalen Fehler in einem dieser Systeme geben würde dies den Zusammenbruch von beinahe der gesamten westlichen Welt bedeuten.
Blue sky als Konzept steht eigentlich nur für eine Applikation die nicht von der Cloud abhängig ist (Cloud → Wolke, ohne Wolken am Himmel gibt es einen blauen Himmel → Blue sky).
Um dies zu erreichen, aber gleichzeitig nicht so ineffizient wie Blockchain zu werden, wird meist peer-to-peer als Architektur gewählt. Dieses mal reden wir allerdings von *Client peer-to-peer*, anders als das obige *Server peer-to-peer*. Wir gehen davon aus, dass sich unsere Geräte und Nutzer direkt miteinander Verbinden. Als gutes Beispiel für dieses Konzept in Aktion ist oftmals von WebRTC[12] die Rede. WebRTC an sich ist unglaublich faszinierend und umfangreich genug um ein eigenes Buch damit zu füllen. Hier wollen wir aber nur die Grundlagen sowie einige Details zur Netzwerkstruktur anschauen.
Tatsächlich ist dies die perfekte Zeit um über WebRTC zu reden. Der Standard ist hauptsächlich dafür verantwortlich, ein einheitliches System zur direkten Webcam- und Mikrofon-Kommunikation zwischen zwei oder mehr Browsern zu erlauben. Da Videochat immer beliebter wurde, aber die Datenmenge es sehr langsam machte jedes Bild über einen Server zu schicken, entwickelte Google WebRTC. Tatsächlich ist es die erste (und einzige) Möglichkeit überhaupt, zwei Browser direkt miteinander zu verbinden. Dafür braucht es momentan lediglich einen *Signaling-Server*, der die Verbindung der beiden Browser aufbaut. Danach läuft die gesamte Kommunikation direkt von einem Browser zum anderen.
Allerdings wurde WebRTC ursprünglich für Video-Konferenzen gebaut, wo-

durch standardmässig alle Video-Streams zu allen anderen Clients geschickt werden. (Es wird also ein `fully connected mesh` aufgebaut). Da dann jeder Client von allen anderen einen Stream empfangen muss und zusätzlich auch noch die eigentlichen Frames decodieren muss, ist `WebRTC` momentan nicht in der Lage mehr als vier oder fünf Verbindungen gleichzeitig zu unterstützen. Tatsächlich ist das auch der Grund wieso `Microsoft Teams` nur vier Bilder gleichzeitig Anzeigt.
Zusätzlich bietet `WebRTC` auf `TURN`[13], `SFU`[14] und `Gateway` Server[15] die es erlauben mehr Verbindungen zu unterstützen (Ähnlich wie `Meet`), aber diese sind nicht Teil dieser Arbeit.

Kapitel 7

Verteilung

Jedes *distributed system* muss gewisse Berechnungen durchführen. Entweder um mit bestehenden Werten neue zu berechnen, oder um eingehende Werte zu speichern (Für die folgenden Beispiele werden wir beide Lösungen gleich behandeln, aber es gibt selbstverständlich Unterschiede). Beim erklären der verschiedenen Architekturen wurde das Thema bereits angesprochen, aber wir müssen und damit beschäftigen, wie wir unsere Prozesse auf verschiedene Nodes verteilen. Am *einfachsten* wäre es natürlich, dem Nutzer diese Wahl zu überlassen, dies ist aber meist nicht möglich oder zu komplex. Auch *zufällige* Lösungen funktionieren nicht sonderlich gut, die Details dazu werden wir gleich noch sehen. Im Abschnitt zu *consistent hashing* werden wir rein zufällige Lösungen noch etwas genauer anschauen, aber es sollte relativ schnell klar werden, dass solche Implementierungen schnell zu Chaos und Komplexität führen.
In diesem Abschnitt wollen wir also einige populäre Lösungen für das Verteilen von Berechnungen anschauen. Es sind nicht die einzigen und wahrscheinlich auch nicht die besten oder einzigen Implementierungen aber einige der populärsten.

7.1 Consistent hashing

Wir haben nun also verschiedene Prozesse und Aufgaben, sowie verschiedene Nodes auf denen diese Prozesse ausgeführt werden sollen. Es stellt sich allerdings die Frage, wie diese Prozesse zu den einzelnen Nodes passen sollen. Die einfachste Lösung wäre dabei natürlich die Entscheidung *zufällig* zu treffen, aber alleine schon damit kommen viele Probleme auf. Tatsächlich gibt es überraschend viele Probleme, eine einfache, zufällige Verteilung zu imple-

mentieren. Eine theoretisch optimale Lösung würde dafür sorgen, dass die Arbeit gleichmässig über das gesamte `Cluster` verteilt durchgeführt wird. Je nach Implementation endet man mit einer ähnlichen Version wie den später besprochenen `Hashring`, nur mit mehr Nachteilen. Tatsächlich löst ein komplett Zufälliges System keine Probleme und macht die Implementierung auch nicht einfacher. Mit etwas Vorwissen kann man den `Hash` von einem genügend komplexen Wert wie beispielsweise der aktuellen Uhrzeit schon als Zufallswert sehen. Daher gibt es keinen Sinn tatsächliche Zufallswerte ohne eine `Hash`-Funktion zu verwenden.
Beispielsweise kann es vorteilhaft sein, gleiche Berechnungen auf der gleichen Node durchzuführen, da die Daten bereits dort sind. Mit einer zufälligen Lösung ist dies aber natürlich unmöglich. Daher ist die einfachste Lösung für das gleichmässige Verteilen ein Hashring, selbst wenn man keine internen Werte wie `ID's`, sondern einfach nur die Zeit nimmt ist es immer noch vorteilhaft.

1. Hashing: Um *Consistent hashing* zu verstehen ist als erstes ein gewisses Grundverständnis zu *Hashing*[10] von Nöten.
   Es gelten einige bedingungen für Hash-Funktionen:

   (a) Für jeden Input gibt es immer genau einen Hash-Wert.

   (b) Die Funktion hat keine Nebenwirkungen und nur einen Input, produzíert also immer den gleichen Hash-Wert.

   (c) Eine kleine Änderung im Ausgangs-Wert sorgt für einen komplett neuen Hash-Wert.

   (d) Hash-Werte haben eine fixe Länge, unabhängig vom Input.

   (e) Von einem Hash-Wert lässt sich der Input nicht einfach wieder herstellen. (Es sind also /Einwegfunktionen)

   (f) Zwar sind Hash-Werte nicht einzigartig, aber durch die vielen Möglichkeiten ist die Wahrscheinlichkeit einer Kollision äusserst gering.

   Als Beispiel kann man sich den Algorithmus `sha1` anschauen:

   ```
   sha1("Message")
   "68f4145fee7dde76afceb910165924ad14cf0d00"
   ```

   Aber nur eine kleine Änderung des Inputs führt zu einem komplett anderen Wert:

```
sha1("Mesage")
"f3ea2a6a072670e2e64460dad44549c5c03efa09"
```

Momentan schauen wir uns nur die *Base64* repräsentierung der Werte an. Natürlich können wir das Ganze auch in *Base10* anschauen, wodurch wir dann Arithmetik durchführen können.

2. Hash-Ring: Die bekannteste Implementierung eines Hash-Rings[11] ist wahrscheinlich `Riak`. Wir werden später, vor allem im Kapitel zu Datenbanken noch `Riak` genauer anschauen, aber momentan fokussieren wir uns einfach auf den verwendeten Hash-Ring. Da man im Falle von `Riak` meist von Datenbanken redet haben wir einen primären Schlüssel mit dem wir unsere Daten speichern und finden können. Wir gehen der einfachheit halber davon aus, dass jeder Eintrag einen einzigartigen Schlüssel besitzt.
   Wenn wir davon ausgehen, dass unser `Cluster` drei Nodes besitzt und den folgenden Wert wollen:

   ```
   {key: "333217a0-99b7-11ea-bb37-0242ac130002", value: "Lorem ipsum"}
   ```

   Um diesen Wert auf einer unserer drei Nodes zu speichern müssen wir als erstes den Hash-Wert des Schlüssels berechnen:

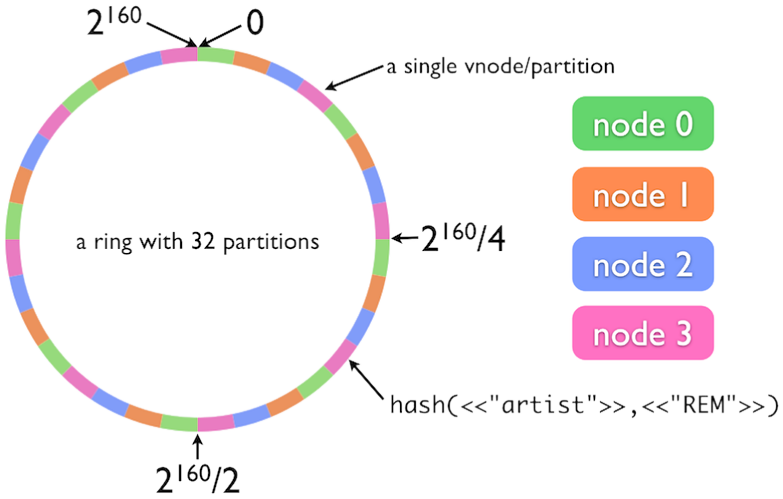
   ```
   sha1("333217a0-99b7-11ea-bb37-0242ac130002")
   "670d5ad675ba3c5899bcb57db719193557e83e93"
   ```

   In einem nächsten Schritt müssen wir dann die *Base10* repräsentierung des Hash-Wertes berechnen. Dafür verwenden wir etwas `Elixir`-Code, den wir später noch genauer analysieren werden.
   Daraus erhalten wir die folgende, sehr grosse Zahl:

   ```
   363730643561643637356261336353839396263...
   ...623537646237313931393335353765383365393 3
   ```

   Da wir nur drei Nodes haben können wir nicht einfach die erhaltene Zahl als Angabe für die Einteilung verwenden. Stattdessen rechnen wir die Zahl *modulo* die Anzahl Nodes, also drei. Als Resultat erhalten wir dann die passende Node.
   $363730643...3833653933 \bmod 3 = 1$
   Nun wissen wir also, dass wir unseren Wert „Lorem ipsum“ auf Node eins zu speichern.

Wenn wir dann später unseren Wert wieder haben wollen, müssen wir lediglich den Schlüssel wieder hashen, um die passende Node finden. Natürlich kommen noch weitere Komplikationen hinzu, zum Beispiel wenn es um das Ausfallen oder Ersetzen von Nodes geht.
Im Falle von `Riak` haben wir natürlich nicht nur drei Nodes und einiges längere Schlüssel. Auch werden Daten oftmals mehrfach gespeichert, um das Ausfallen von einzelnen Nodes weniger schlimm zu machen.
`Riak` wird oftmals mit der folgenden Grafik erklärt:

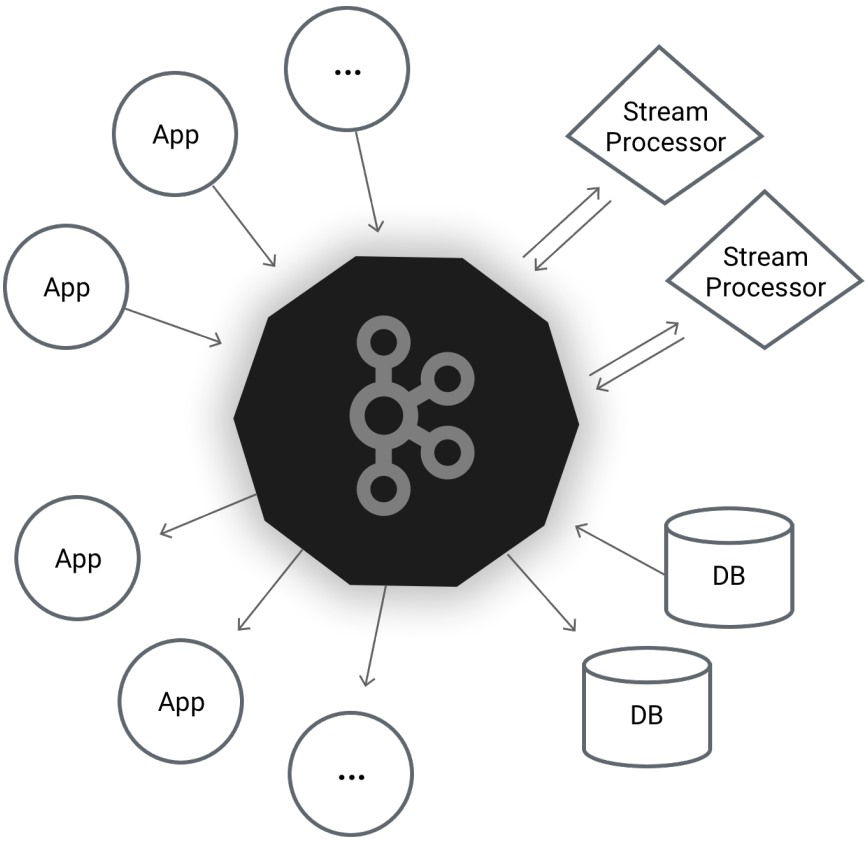


`Riak` teilt den Ring in verschiedene Stücke auf, die dann von den Nodes verwaltet werden. Zusätzlich hat `Riak` noch das Konzept von `VNodes`, die das dynamische ändern der Ringstruktur einfacher machen. Zwar kann `Riak`-Core vielseitig eingesetzt werden, wird die Architektur am häufigsten für `Dynamo-Style` Datenbanken verwendet.

7.2 Apache Kafka

Wenn bereits einige Vorkenntnisse vorhanden sind, kann man sich `Apache Kafka` als eine Mischung aus PubSub und Task Queue vorstellen.
Auch wenn es inzwischen verschiedene Open- und Closed-Source implementierungen der `Kafka`-Architektur existieren, ist `Kafka` selbst weiterhin die beliebteste Variante. Ursprünglich kam das Projekt von LinkedIn, wurde aber dann veröffentlicht und ist jetzt Teil der `Apache Software Foundation`.

`Kafka` lässt sich wahrscheinlich am besten mit der folgenden Grafik[4] erklären:



Wir werden `Kafka` selbst momentan einfach als eine einzelne schwarze Box vorstellen und uns nicht mit den Innereien befassen.
Jeder unserer Clients, die Daten für das System haben können diese mitsamt Instruktionen an `Kafka` schicken. Beliebige andere (*Consumer*) Nodes können dann die Daten von `Kafka` nehmen. Dadurch ist es an den *Consumer*-Nodes zu entscheiden, welche Daten für ihre Situation passend sind. Natürlich ist diese Erklärung extrem vereinfacht und erklärt mehr die Interaktion mit `Kafka` als die Funktionsweise selbst. `Kafka` funktioniert also besser, wenn es viele verschiedene Quellen und Konsumenten von Daten gibt, während *Consistent hashing* wahrscheinlich die bessere Option für ein uniformes Cluster, bei dem jede Node ähnliche Aufgaben übernimmt.

Kapitel 8

Compute

Bisher haben wir *Storage* und *Compute*[1] als ein einziges Gebiet behandelt. Auch wenn jede *Compute* Applikation auch gewisse *Storage* Elemente benötigt, lassen sich die meisten Probleme mit *distributed systems* trotzdem mehrheitlich auf *Storage* zurückführen. Denn für die Systeme spielt es schlussendlich keine Rolle ob die Daten auf eine Festplatte geschrieben werden oder nur Berechnungen und Veränderungen durchgeführt werden. Selbst das auf die Festplatte schreiben kann schon als Berechnung gesehen werden. In diesem Abschnitt fokussieren wir uns allerdings auf Systeme, die grössere Datenmengen einnehmen und ausgeben, wobei viel `CPU` Zeit dafür benötigt wird.

8.1 Riak-Core

Im Kapitel zur Verteilung haben wir `Riak` bereits angeschaut. Es ist tatsächlich überraschend einfach, mit `Riak`, beziehungsweise dem Kern des Projektes, `Riak-Core`, ein gutes Verteilsystem zu implementieren.
`Riak-Core` entstand durch die reorganisation der `Riak` Datenbank. Da neben dem *normalen* `Riak-KV` (Key-Value Store) auch zusätzlich `Riak-TS` (Time-Series) und `Riak-Cloud-Storage` (Object Storage) dazu kamen. Alle bauen auf `Riak` auf, aber es war mühsam, jedes mal die *unnötigen* Teile zu entfernen. Daher wurden die wichtigsten Teile mit `Riak-Core` zu einem eigenen Projekt.
`Riak-Core` verwendet ein Komponenten-System. Dabei gibt es zwei relevante Komponenten die wir hier noch genauer anschauen werden, wobei wir auch schon den ersten Blick auf `Erlang` und `Elixir` Code werfen, den wir hoffentlich mit späteren Kapiteln noch genauer verstehen werden.

Für diese Erklärung werden wir das Pingring Beispiel von Ben Tyler verwenden.

1. Service: Der `Service` ist die `API` mit der man mit der Applikation interagieren kann. Als Beispiel könnte der Code dafür wie folgt aussehen:

```
defmodule Pingring.Service do
        def ping do
                doc_idx = :riak_core_util.chash_key(
                 {
                         "ping",
                         :erlang.term_to_binary(:os.timestamp())
                 }
                )
                pref_list = :riak_core_apl.get_primary_apl(
                        doc_idx,
                        1,
                        Pingring.Service
                )
                [{index_node, _type}] = pref_list
                :riak_core_vnode_master.sync_spawn_command(
                        index_node,
                        :ping,
                        Pingring.Vnode_master
                )
        end
end
```

   Momentan wird nur eine einzige Funktion (`ping/0`) implementiert, aber natürlich ist es möglich weitere hinzu zu fügen. Als erstes erhalten wir einen Key. Natürlich sollte eine Datenbank einen Wert der Inputs hashen aber für dieses Beispiel reicht es einfach die aktuelle Zeit. Dann fragen wir nach der aktuellen `APL` (Active preference list) für unseren Key. Die `APL` sagt aus, an welche Nodes die Anfrage geschickt werden soll. Da wir nur einen Ping schicken, wollen wir nur eine Node erhalten. Daher ist der zweite Parameter für die Funktion `1`. Zum Schluss müssen wir auch noch `Service` angeben, was aber nur relevant wird sobald mehrere vorhanden sind. Danach können wir die Anfrage schicken, in unserem Falle mit dem `:ping` Befehl.

2. VNode: Mit dem Service können wir einen Befehl an das System schicken. Dieser muss dann allerdings von Code in einer `VNode` verarbeitet werden. Dabei muss sich der Programmierer nicht um die Verteilung oder Concurrency kümmern. Lediglich die beiden Komponenten müssen vorhanden sein.

```
defmodule Pingring.Vnode do
        require Logger
        @behaviour :riak_core_vnode

        def start_vnode(partition) do
                :riak_core_vnode_master.get_vnode_pid(
                partition,
                __MODULE__
                )
        end

        def init([partition]) do
                {:ok, %{partition: partition}}
        end

        def handle_command(
                :ping,
                _sender,
                %{partition: partition} = state) do
                Logger.warn("got a ping request!")
                {:reply, {:pong, partition}, state}
        end

        def handle_handoff_command(_fold_req, _sender, state) do
                {:noreply, state}
        end

        def handoff_starting(_target_node, state) do
                {true, state}
        end

        def handoff_cancelled(state) do
                {:ok, state}
        end
```

```
        def handoff_finished(_target_node, state) do
            {:ok, state}
        end

        def handle_handoff_data(data, state) do
            {:reply, :ok, state}
        end

        def encode_handoff_item(object_name, object_value) do
            ""
        end

        def is_empty(state) do
            {true, state}
        end

        def delete(state) do
            {:ok, state}
        end

        def handle_coverage(req, key_spaces, sender, state) do
            {:stop, :not_implemented, state}
        end

        def handle_exit(pid, reason, state) do
            {:noreply, state}
        end

        def terminate(reason, state) do
            :ok
        end
end
```

Auf den ersten Blick wirkt die `VNode` überwältigend. Aber da wir uns nur um die `:ping` Funktion kümmern können wir den restlichen Code für den Moment weglassen. Dann sieht die `VNode` also wie folgt aus:

```
defmodule Pingring.Vnode do
        require Logger
        @behaviour :riak_core_vnode
```

```
        def handle_command(
                :ping,
                _sender,
                %{partition: partition} = state) do
                Logger.warn("got a ping request!")
                {:reply, {:pong, partition}, state}
        end
end
```

In einem späteren Kapitel werden wir uns noch genauer mit `GenServern` auseinandersetzten. Der dort verwendete Syntax sollte an diesen erinnern. In *sinnvollen* Applikationen gäbe es natürlich verschiedene Parameter an die Funktion. Für dieses Beispiel senden wir aber einfach die Partition in der die Anfrage verarbeitet zurück.
Schlussendlich braucht es nur zwei Funktionen um ein funktionsfähiges `Riak-Core` Cluster zu bauen (Natürlich kommen noch einige weniger wichtige Funktionen hinzu, die man allerdings meist kopieren kann).

8.2 Sonderfälle

Im Fokus dieser Arbeit stehen meist *distributed systems* und Cluster die mit dem Internet interagieren und viele, gleichzeitig laufende Prozesse verarbeiten. Aber es gibt auch andere Nutzen für Cluster von Prozessoren und Computern.

1. Machine learning: Im Jahre `1956`, relativ kurz nach der *Erfindung* der ersten Computer, war meist das Speichern von Daten der teuerste Aspekt. Während wir heute von `Gigabytes`, `Terabytes` oder manchmal sogar `Petabytes` reden waren Festplatten damals unglaublich teuer und gross. Damals war es ein Grossunternehmen `5 Megabytes`[18] zu verschicken.

Inzwischen haben Geräte in unseren Taschen das vielfache dieser Kapazität. Aber meist steigt die tatsächliche Kapazität für Objekte nicht gleichmässig an, da wir uns angewöhnen keine Daten mehr zu löschen oder zu archivieren. Über die letzten Jahre wurden wir alle zu Datenhordern, meist ohne es zu merken. Auch wenn es noch am oberen Ende des Spektrums liegt, ist es heute nicht mehr besonders überraschend die Installation von mehreren `petabyte` Speicher zu beobachten.
Aber nicht nur Einzelpersonen haben mehr Daten. Auch (oder besonders) Firmen besitzen mehr Daten über ihre Nutzer als jemals in der Vergangenheit [22]. Um die Produkte zu optimieren und die Erfahrung der Nutzer zu verbessern setzten viele dieser Firmen auf `Machine learning`. Auch dieses Thema ist unglaublich komplex und viel zu gross um auch nur in dieser Arbeit an zu schneiden.
Es ist aber wichtig zu verstehen, dass das verarbeiten dieser unglaublichen Datenmengen enorme Cluster an Computern benötigt. Allerdings ist die Situation anders als bei den obigen Beispielen. Meistens werden Modelle im voraus trainiert und erst später an die eigentlichen Nutzer geliefert. Auf solchen Clustern läuft also selten Code von dem Nutzer in Echtzeit abhängen. Daher ist es meist Akzeptabel einen Controller oder Master zu haben, der die restlichen Berechnungen überwacht und Koordiniert, da Ausfälle weniger verheerend sind.
Aber auch die effiziente Berechnung mit grossen Datensets ist ein unglaublich komplexes Thema und nicht der Fokus dieser Arbeit.

2. Rendering: Wer schon mal einen Film geschnitten hat oder sich mit Videobearbeitung auseinandersetzt kennt das Problem der langen Renderzeiten. Dabei geht es meist nur um kleine Hobby Projekte oder

Ferien-highlights. Aber für grössere Produktionen wird es nur noch extremer. Beispielsweise der Disney Film `Toy Story 3` brauchte bis zu 39 Stunden Zeit für ein einziges Frame. Für einen Kinofilm mit 24 Bildern pro Sekunde waren also Wochen von nöten. Und das obwohl Disney zwei grosse Renderfarmen (Datencenter mit tausenden von Servern besonders für das Rendern von Filmen gebaut) verwendet. Ähnlich wie die Systeme für `Machine learning` wird auch hierfür meist ein zentralisiertes System verwendet, da die Auswirkungen kleinerer Ausfälle minimal gehalten werden können.

Kapitel 9

Storage

Oftmals wird weniger über *distributed systems* im Zusammenhang mit Datenbanken und Speichersystemen nachgedacht, da die wenigsten diese selber bauen. Für beinahe alle Projekte wird eine bewährte Lösung gewählt und es wird sich nicht gross Gedanken über deren interne Probleme gemacht.
Da allerdings, wie oben bereits erwähnt, viele der Probleme mit *distributed systems* im Zusammenhang mit Zuständen und Speicher entstehen, ist es nicht überraschend dass Datenbanken am stärksten gegen diese Probleme kämpfen.

9.1 Dynamo

Während es hunderte von verschiedenen Datenbanken gibt lassen sich diese doch in einige Kategorien einteilen.

- SQL: Datenbanken die einem klaren Schema folgen sind einiges einfacher zu durchsuchen und es ist meist weniger aufwendig mit ihnen zu interagieren. Auch unterstützden SQL Datenbanken auch meist das Verknüpfen von verschiedenen Werten und Schemas. Allerdings sind solche meist auf eine einzige Maschine beschränkt, da sonts die Schemas nicht gut durchgesetzt werden können, oder die Suchanfragen unglaublich ineffizient über verschiedene Nodes verarbeitet werden. Auch wenn es möglich ist, kommt es äusserst selten vor.

- NoSQL: Der Name ist Programm. Für solche Datenbanken gibt es meist kein klares Schema und viele implementieren ihre eigene `Query` Sprache. Auch werden oft komplette `JSON` Objekte gespeichert. Diese

Eigenschaft wird vor allem vorteilhaft wenn die Daten sehr unregelmässig sind oder keinem klaren Schema folgen. Die genauen Unterschiede und Vor- oder Nachteile der beiden Systeme sind eine Arbeit für sich und diese dann in ein *distributed system* zu verwandeln braucht schon einige Seiten Text. Zum Glück hat genau das `Amazon` bereits erledigt. Mit dem Dynamo Paper[17] veröffentlichte die Firma den defacto Standard für *distributed Datenbanken.*

Dynamo selbst ist die Datenbank welche Amazon intern über Jahre entwickelte und sie ist heute noch für den Online Store sowie viele `AWS` Dienste wie beispielsweise `S3` verantwortlich. Wer heute noch selbst eine *distributed datenbank* entwickeln will folgt meist diesen Regeln und Standards oder wählt einige davon aus. Datenbanken die den Dynamo-Prinzipien stark folgen werden oftmals als *Dynamo-Architektur* oder *Dynamo-Style* bezeichnet.

9.2 Riak

Die verschiedenen Projekte und Produkte rund um `Riak` wurden schon zwei mal in dieser Arbeit besprochen. `Riak` selbst ist in verschiedenen Datenbank-Arten vorhanden, wobei alle dem originalen *Dynamo paper* sehr treu folgen. Einer der grössten *Nachteile* (Und gleichzeitig einer der grössten Vorteile) der Datenbank ist die Wahl der Programmiersprache. Während `Erlang` das Entwickeln solcher Systeme unglaublich einfach und sicher macht, ist `Riak` momentan auf etwa 65 Nodes pro Cluster beschränkt. Für die allermeisten Nutzer ist dies genügend, aber leider gibt es immer Ausnahmen die mehr von ihren Datenbanken verlangen.

9.3 Cassandra

Anders als `Riak` folgt `Cassandra` dem *Dynamo paper* weniger treu. Auch setzt diese Datenbank nicht auf `Erlang` sondern auf `Java` als Programmiersprache. In vielerlei hinsicht ist `Cassandra` `Riak` überlegen. Beispielsweise macht es die Architektur der Datenbank relativ einfach im Laufe der Zeit weitere Nodes zum Cluster hinzu zu fügen. Auch zeigt sich `Cassandra` in vielen Situationen schneller und effizienter. Dazu kommt noch, dass `Cassandra` Cluster weit über 65 Nodes hinaus wachsen können. Aber es muss natürlich erwähnt werden, dass die beiden Systeme auf verschiedene Aspekte ihren Fokus setzen. `Riak` versucht möglichst vorhersehbar und Ausfallsicher zu

sein. Dadurch wurde Riak auch zur bevorzugten Datenbank für die NHS die unter keinerlei Umständen Daten verlieren dürfen oder auch nur für kürzeste Zeit die Datenbank offline nehmen dürfen.
Anders setzt Cassandra darauf, möglich grosse Datenmengen und möglichst viele Anfragen gleichzeitig zu verarbeiten, wodurch es die Datenbank der Wahl für Netflix und Apple wurde.
Leider ist Cassandra ein unglaublich Komplexes Thema und verdient eigentlich ein eigenes Kapitel oder eine eigene Arbeit. Cassandra ist sehr effizient für grosse Datenmengen, aber durch die Wahl der Programmiersprache und die Probleme der JVM gibt es trotzdem noch Spielraum. Daher begann die Entwicklung eines neuen Projektes namens Scylla-DB. Diese neue Datenbank ist kompatibel mit Cassandra und verwendet die gleichen Protokolle und Standards, ist allerdings in C++ geschrieben, wodurch das System einiges effizienter wird.
Für weitere Informationen über Cassandra lohnt es sich die offizielle Dokumentation[25] zu lesen oder auch einfach die Datenbank zu testen.

Teil III

Programmierung

Nach den theoretischen Grundlagen und Konzepten im Zusammenhang mit *distributed systems* kennen gelernt haben, ist es nun an der Zeit sich mit der Programmierung zu befassen. Aber ähnlich wie die Theorie ist auch die praktische Anwendung voller Fallen und Problemen mit denen man sich auseinandersetzen muss. Auch ist wichtig zu verstehen dass die Programmierung von *distributed systems* natürlich auf der Theorie aufbaut. Daher werden die oben besprochenen Regeln weiterhin gelten und das Missachten dieser kann schlimme Folgen haben.
Um das Problem noch zu verschlimmern unterstützen die meisten Programmiersprachen nur sehr bedingt die Entwicklung von solchen *distributed system*. Natürlich ist es mit allen *Turing-Complete*-Programmiersprachen möglich, aber manche liefern die nötigen `API's` und Frameworks mit, während man für andere viel Zeit aufwenden muss nur um überhaupt anfangen zu können. Daher ist es schneller eine neue Programmiersprache zu lernen, die mit dem Gedanken an solche Probleme gebaut wurde.
Wir haben bereits etwas `Elixir` Code angeschaut, müssen aber als erstes `Erlang` besprechen, welches als Grundlage für `Elixir` notwendig ist. Dazu kommen dann noch die genialen Tools die direkt mit `Erlang` und `OTP` mit geliefert werden und die es sehr einfach machen *distributed systems* zu schreiben in dem sie viele der grundlegenden Probleme abstrahieren oder sich direkt selbst darum kümmern.

Kapitel 10

Erlang

`Erlang` wurde zum ersten mal im Jahre 1987 veröffentlicht. Das Telefon Unternehmen `Ericsson` war damals weltweit führend wenn es um Telefonie ging. Aber anders als moderne Internet Dienste gab (und gibt es immer noch) besondere Probleme und Erwartungen an Telefonsysteme. Während es heute noch mehr oder weniger akzeptabel ist, dass eine Webseite für einige Minuten offline geht, um Wartungen durchzuführen ist es für ein Telefonsystem einfach nicht akzeptabel. Und trotzdem müssen Telefonsysteme aktualisiert werden. Es muss also möglich sein, neue Funktionen hinzuzufügen und Probleme zu beheben, ohne dass die Nutzer etwas davon mitbekommen.
Die Firma entwickelte dann `Erlang` um genau diese Probleme anzugehen. Inzwischen ist die Sprache Open-Source und für viel mehr als nur Telefonie einsetzbar. Für die Entstehung `Erlangs` ist es ebenfalls wichtig zu verstehen, wie die Hardware der Telefon-Infrastruktur funktioniert:

- Eine vielzahl verschiedener Telefonanschlüsse sind mit einem Switch verbunden.
- Ein Switch besteht aus vielen einzelnen Computern, die mit einem physikalischen Backplane verbunden sind.
- Ein solches Backplane hat beinahe unlimitiert Bandbreite.
- Über jeden Switch laufen viele Telefonanrufe gleichzeitig.

Ursprünglich war die Software für diese Telefonie-Systeme in `C` geschrieben. Aber es wurde schnell klar wie umständlich und unnötig kompliziert `C`-Threads für das Verwalten von hunderten von Telefonverbindungen ist. Also traf `Ericsson` die Entscheidung den extra Aufwand zu akzeptieren

und eine Software mit dazugehöriger Programmiersprache zu entwickeln, die das schreiben von Telefonie Anwendungen einfacher machen sollte. Daraus wurden dann `Erlang` und `OTP`.
`Erlang` wurde immer mehr erweitert und neue Funktionen hinzugefügt. Auch wenn die Sprache anfangs nur für `Ericsson` Telefon-Switches designed war, wurde die Sprache dann veröffentlicht und für weitere Plattformen herausgegeben.
Man kann auch nicht leugnen, dass während dieser Zeit wurde die Gemeinschaft um `Erlang` auch etwas Arrogant wurden. Es ist kaum umstritten dass `Erlang` eine der besten `Concurrency` Implementierungen überhaupt hat, trotzdem ist es nicht die einzige. Aber (leider) gibt es das bekannte Zitat von einem der Ersteller der Sprache, Robert Virding:

> Any sufficiently complicated concurrent program in another language contains an ad hoc informally-specified bug-ridden slow implementation of half of Erlang. – Robert Virding[23]

(Zu Deutsch: Jedes ausreichend komplizierte nebenläufige Programm in einer anderen Sprache enthält eine ad hoc informell spezifizierte, fehlerbehaftete, langsame Implementierung von Erlang.)
Dies mag vielleicht überspitzt klingen, aber tatsächlich ist einige Wahrheit dahinter. Auch heute noch ist `Erlang` die dominierende Programmiersprache für Telefonie Systeme Weltweit und noch immer verantwortlich für eine mehrheit der Weltweiten Anrufe sowie Mobiler-Webtraffic.

10.1 Syntax

Auch wenn `Erlang` die Grundlage für `Elixir` ist, soll in dieser Arbeit der Fokus nur auf `Elixir` liegen, da jede Funktion und `API` von beiden Sprachen beinahe identisch erreichbar ist.
Trotzdem wollen wir kurz einige Beispiele für den Syntax anschauen:
Als erstes ein kleines Beispiel von *Wikipedia* für eine rekursive Fakultätsberechnung:

```
-module(fact).
-export([fac/1]).

fac(0) -> 1;
fac(N) when N > 0, is_integer(N) -> N * fac(N-1).
```

Der Syntax wirkt auf den ersten Blick sehr ungewohnt und ist kaum mit einer anderen Sprache zu vergleichen. Aber wenn man sich an die Funktionalität

gewöhnt und sich auf die Erlang-Art einlässt, kann man mit wenigen Zeilen Code hunderte Zeilen C-Code ersetzen.
Beispielsweise Fibonacci:

```
-module(series).
-export([fib/1]).

fib(0) -> 0;
fib(N) when N < 0 -> err_neg_val;
fib(N) when N < 3 -> 1;
fib(N) -> fib_int(N, 0, 1).

fib_int(1, _, B) -> B;
fib_int(N, A, B) -> fib_int(N-1, B, A+B).
```

Als gutes Beispiel für die Fähigkeiten der Sprache ist Erlang: The Movie zu empfehlen.

10.2 Let it crash

Wir werden später noch *Supervisors* genauer anschauen, aber für den Moment wollen wir uns die Mentalität für Erlang Entwickler anschauen.
Auch wenn `let it crash` auf den ersten Blick äusserst kontraproduktiv wirkt, verbirgt sich viel Sinn und Überlegung in der einfachen Aussage. Jeder normale Mensch würde, wenn gefragt, sagen dass Abstürze um jeden Preis verhindert oder zumindest minimiert werden sollen. Aber tatsächlich ist es Vorteilhaft Abstürze zu akzeptieren, solange dies in einer kontrollierten Art und im Kleinen geschehen.
Ein *Supervisor* erlaubt es, einzelne Prozesse unabhängig von anderen Neuzustarten. Da wir (*hoffentlich*) auch das restliche System in einer unabhängigen Art gebaut haben können wir Prozesse unabhängig Neustarten und Ausfälle isolieren.
Die Erlang Ideologie besagt also, anstatt viel Rechenleistung mit dem Lösen von Problemen zu verschwenden soll der Prozess der für das Problem verantwortlich ist einfach abstürzen und von neuem Anfangen. Natürlich geht dass nicht immer, aber in vielen Situation ist es einfacher als hunderte von besonderen Sonderfällen und Ausnahmen von Hand zu überwachen.
Aber es ist wichtig zu verstehen, dass man nur von dieser Eigenschaft profitieren kann wenn man sein *distributed system* von Anfang an in einer unabhängigen Art gebaut hat.

10.3 Distributed Erlang

Im nächsten Kapitel schauen wir uns `Elixir` an, welches eine Sprache mit ähnlichen Funktionen aber angenehmeren Syntax ist, daher werden wir uns auch *distributed Erlang* in `Elixir` vertraut machen. Die tatsächlichen Mechaniken und Funktionen sind meist beinahe identisch, nur der Syntax ist etwas verschieden.
Wir haben bereits verschiedene Cluster-Strukturen angeschaut. Es ist wichtig zu verstehen dass `Erlang` standardmässig nur ein `fully connected mesh` unterstützt. Um mit einer Node einem Cluster beizutreten brauchen wir lediglich eine einzige Funktion aus dem Node Modul.
Mit `Node.connect/1` verbinden wir eine Node mit einer anderen. Dabei ist es egal, ob eine der beiden Nodes bereits Teil eines Clusters ist. Da `Erlang` damit rechnet ein `fully connected mesh` aufzusetzen kümmert sich die virtual machine automatisch darum, alle Nodes mit allen anderen zu verbinden. Zum finden und verbinden zweier Nodes verwendet `Erlang` ein eigenes `DNS` ähnliches System, mit welchem jeder Node beim starten ein Name gegeben wird. Normalerweise folgen diese dem Muster

$$node\_name@host\_name$$

oder

$$node\_name@ip\_address$$

Es ist auch wichtig zu verstehen dass dieses System nur zum verbinden benutzt wird. Danach muss man so gut wie nie mehr auf diese Informationen zu greifen. Sobald eine Node teil eines Clusters ist (Auch nur zwei verbunden Nodes sollen hier schon als Cluster gezählt werden) sind alle öffentlich registrierten Prozesse von allen Nodes im Cluster erreichbar, ohne dass man sich gross Gedanken darüber machen muss.

1. Partisan: Wir haben bereits die Vor- und Nachteile eines `fully connected mesh` angeschaut. Auch `Erlang` verwendet ein `Heartbeat protocol` um die Nodes im Cluster zu erkennen. Dadurch, und durch den Fakt dass die Anzahl der Verbindungen im Quadrat mit der Anzahl Nodes steigt, ist `Erlang` nur bedingt skalierbar. Von den Entwicklern selbst wird eine maximalgrösse von etwa 50 - 75 pro Cluster empfohlen. Zwar gab es bereits erfolgreiche Tests mit bis zu 500 Nodes, aber nie ohne dabei auf grosse Probleme zu treffen. Allerdings implementiert die Sprache selbst keinerlei Cluster-Struktur (Sondern Module wie Node und die *send* Funktionen) und es ist theoretisch möglich, eine eigene

hinzuzufügen. Genau das ist das Ziel einiger Tests und Untersuchungen die seit einigen Jahren durch, von der EU unterstützte, Wissenschaftler an verschiedenen europäischen Universitäten durchgeführt werden.
Mit Partisan haben diese eine Library, die es Nutzern erlaubt eigene Layouts für ihre Cluster zu wählen. Unterstützt sind momentan Client-Server und P2P. Diese kommen natürlich mit allen bereits angesprochenen Vor- und Nachteilen und sind überraschend einfach zu implementieren. So muss man lediglich die Funktionen, die von der Cluster-Struktur abhängig sind, wie beispielsweise send/2 durch die Partisan Äquivalente ersetzen.

2. Libcluster: Da immer mehr Deployments über Docker ablaufen gibt es immer mehr Probleme für Erlang und Elixir, da die Clustering Lösungen der beiden Sprachen nicht zu den Standards für Docker passen. Auch ist es unglaublich umständlich jede neue Node von Hand zum Cluster hinzuzufügen und sich um Ausfälle zu kümmern. Da viele bereits ähnliche Probleme angetroffen hatten und das Problem immer klarer wurde, begann die Suche nach einem einfacheren, automatischen Weg. Nach einigen Fehlversuchen entstand dann libcluster. Mit dieser Library ist es sehr einfach dynamische Cluster mit Docker zu bauen. Die Library unterstützt sowohl Kubernetes, Rancher und die Erlang eigenen Cluster Lösungen. Libcluster unterstützt sogar Partisan.

Kapitel 11

Elixir

In einem vorherigen Kapitel haben wir bereits Erlang und OTP angesprochen. Allerdings wird relativ schnell klar, dass die Sprache ursprünglich nur für Telecom-Applikationen geschrieben wurde. Der Syntax ist (für heutige Standards) sehr ungewohnt und viele der Funktionen mit denen man bei neueren Sprachen fest rechnet sind nicht vorhanden. Vor Allem das Ökosystem und die Community um die Sprache herum lässt oftmals zu wünschen übrig. Aber die Fähigkeiten und Möglichkeiten der Sprache wenn es um Fehlertoleranz und Concurrency geht sind auch nach 30 Jahren noch unübertroffen. Die Virtual-Machine und besonders OTP bieten eine unglaubliche Menge an Funktionen die sich perfekt auf moderne Programme vor allem solche im Zusammenhang mit dem Internet eignen.

Zum Glück haben bereits viele das gleiche Problem angetroffen, woraus dann die Programmiersprache Elixir entstand. Um das Design der Sprache richtig verstehen zu können, müssen wir als erstes die Entstehung und Ursprünge der Sprache anschauen. Im Jahre 1995 veröffentlichte Yukihiro Matsumoto die erste Version von Ruby[26]. Auch wenn die Sprache anfangs kein grosser Erfolg war, kam durch die veröffentlichung von Ruby on Rails Wind in die Segel. Das Framework erlaubte die einfache und schnelle Entwicklung von Webseiten und Servern mit der Ruby Programmiersprache. Dabei lag ein klarer Fokus auf der Entwicklungserfahrung und Geschwindigkeit. Mit Rails war es möglich, in kürzester Zeit komplexe Programme oder Prototypen zu schreiben und auf diesen weiter aufzubauen. Das Framework, aber auch die Sprache, wurden immer populärer, mit einer immer grösseren und sehr treuen Community.
Allerdings kamen im Laufe der Zeit immer mehr Probleme auf. Vor Allem

dadurch, dass Seiten wie Twitter, die ursprünglich auf Ruby on Rails gesetzt hatten, immer beliebter wurden und Computer aber vor allem Server immer mehr Prozessoren bekamen, wurde schnell klar, dass die Sprache nicht mithalten würde. Denn Ruby an sich konnte Anfangs nur einen einzigen Prozessor ausnutzen.
José Valim, ein Ruby-Core Team Mitglied und Entwickler vieler beliebter Rails Libraries, regte sich über die grosse Komplexität und den vielen Aufwand auf, den es benötigte um Ruby Programme schnell und effizient zu machen, vor allem wenn es um mehrere Prozessoren ging. Er traf dann auf Erlang und OTP. Auch wenn er schnell die Vorteile verstand, so realisierte er auch, dass der Syntax zu kompliziert war um für Leute wie die Ruby Community interessant zu werden. Er entschied sich also seine eigene Sprache zu erstellen. Dafür wollte er weiterhin die Erlang Virtual-Machine verwenden um Zugang zu den unzähligen Vorteilen zu bekommen. Aber für den Syntax wollte er sich an Ruby orientieren. Auch sollte ein grosser Fokus auf der Entwicklungserfahrung und Geschwindigkeit liegen. Daraus wurde dann Elixir.

Um den Umfang dieser Arbeit unter kontrolle zu halten, wurde die Entscheidung getroffen, die Sprache Elixir mit Hilfe der offiziellen Dokumentation auf elixirschool.com[19] zu erklären.
Als erstes muss aber einer der grossen Vorteile für das Debugging und Entwickeln von Elixir Programmen erwähnt werden: Die interaktive Konsole. Sie erlaubt es, puren Elixir Code direkt in einer Konsole, ähnlich wie in Python auszuführen. Aber zusätzlich ist es auch möglich, diese in die Umgebung eines existierenden Projektes hinein zu starten, wodurch man dann Zugang zu den dort definierten Modulen und Funktionen erhält.
Sobald Elixir und Erlang auf einem System installiert sind, ist es möglich, die Konsole mit dem Befehl

```
$ iex
```

zu starten (iex steht für interactive Elixir). Für die Einführung in die Sprache ist die Konsole mehr als genügend, da man sich dadurch noch nicht mit Programmen wie dem build-tool oder Package-manager herumschlagen muss.

11.1 Grundlagen

Als erstes ist es wichtig zu verstehen, dass sowohl Elixir als auch Erlang Funktionale Programmiersprachen sind. Anders als beispielsweise

`Python` oder `Java`, die oft als `Objektorientierte Programmiersprachen` bezeichnet werden.

11.2 Funktionales Programmieren

Als Basis einer funktionalen Sprache dient natürlich die Funktion. Meist ist eine Funktion selbst ein Datentyp, kann also beispielsweise als Parameter einer anderen Funktion weitergegeben werden. Auch folgen viele dieser Sprachen dem Konzept, dass Funktionen keine Nebeneffekte haben sollen. Dies hat eine vielzahl an Vorteilen:

- Da eine Funktion nur eine gewisse Anzahl Inputs und Outputs hat, und die Outputs direkt von den Inputs abhängig sind, wird die Funktion zu jedem Zeitpunkt und in jeder Situation mit gleichen Inputs die gleichen Outputs liefern. Dies ist ähnlich wie eine mathematische Funktion. Beispielsweise die Funktion

  $$fn : x \to x * x$$

  wird unter keinerlei Umständen nicht das Quadrat einer Zahl, beispielsweise `42` liefern. Egal ob man die Rechnung dazu im Kopf, auf einem Mobilgerät oder einem Supercomputer ausführt, egal ob man diese Funktion vorher bereits aufgerufen hat oder nicht und egal ob die Variable `x` ausserhalb der Funktion als etwas anderes definiert wurde, die Funktion $fn(42)$ wird immer $1'764$ zurück geben.

- Eine Funktion ohne Nebeneffekte, kann unendlich parallelisiert werden. Solange eine Funktion auf einem Prozessor ausgeführt werden kann, ist es egal wie viele Prozessoren und Funktionen man nebeneinander am laufen hat. Da alle unabhängig voneinander laufen, spielt es keine Rolle. Dieses Thema wurde vorher bereits einmal zum Thema Nebenläufigkeit und Parallelisierung angesprochen.

- Funktionen die dieser Regel folgen sind sehr einfach zu testen. Da bereits bekannt ist, dass die Funktion keine Seiteneffekte hat, muss man zur verifizierung einer Funktion lediglich die Outputs mit den erhofften Outputs mit den gleichen Parametern vergleichen.

Allerdings ist es auch wichtig zu verstehen, dass eine Sprache ohne Nebeneffekte so gut wie Sinnlos ist. So ist beinahe alles, was wir als nützlich in einem Programm bezeichnen würden ein Nebeneffekt. Sei es das Anzeigen

auf dem Bildschirm oder das Speichern in der Datenbank. Daher versuchen die meisten funktionalen Sprachen, wie Elixir, Nebeneffekte nicht unmöglich zu machen, sondern wenn immer möglich davon abzuraten und die notwendigen Nebeneffekte gut erkennbar zu machen.

11.3 Datentypen

Sobald die `iex` Konsole offen ist, kann man bereits mit der Sprache spielen und ohne besondere Fachkenntnisse Code ausführen.

```
iex> 2+3
5
iex> 2+3 == 5
true
iex> String.length("g2IGUnt+")
8
```

Integer:

```
iex> 255
255
```

Unterstützung für Binär-, Oktal- und Hexadezimalzahlen wird mitgeliefert:

```
iex> 0b0110
6
iex> 0o644
420
iex> 0x1F
31
```

Gleitkommazahlen:
In Elixir verlangen Gleitkommazahlen mindestens einen Punkt nach einer Zahl; sie haben 64 Bit double precision und überstützen e für Exponenten:

```
iex> 3.14
3.14
iex> .14
**(SyntaxError) iex:2: syntax error before: '.'
iex> 1.0e-10
1.0e-10
```

Booleans:
Elixir unterstützt `true` und `false` als Booleans; alles außer `false` und `nil` wird als wahr betrachtet:

```
iex> true
true
iex> false
false
```

Atoms:
Ein Atom ist eine Konstante, bei der der Name auch den Wert darstellt. Falls du mit Ruby vertraut bist kennst du Atoms als Symbole:

```
iex> :foo
:foo
iex> :foo == :bar
false
```

Booleans `true` und `false` sind gleichwertig zu den Atoms `:true` und `:false`.

```
iex> is_atom(true)
true
iex> is_boolean(:true)
true
iex> :true === true
true
```

Modulnamen in Elixir sind auch Atoms. MyApp.MyModule ist ein gültiges Atom, auch wenn dieses Modul noch nicht deklariert wurde.

```
iex> is_atom(MyApp.MyModule)
true
```

Atoms werden auch dazu genutzt, um Module aus `Erlang` Bibliotheken zu referenzieren. Dies gilt auch für in Erlang bereits vorhandenen Bibliotheken.

```
iex> :crypto.strong_rand_bytes 3
<<23, 104, 108>>
```

Strings:
Strings in Elixir sind UTF-8 codiert und in doppelten Anführungszeichen zu schreiben:

```
iex> "Hello"
"Hello"
```

```
iex> "d z i  k u j "
"d z i  k u j "
```

Strings unterstützen Zeilenumbrüche und Escapesequenzen:

```
iex> "foo
...> bar"
"foo\nbar"
iex> "foo\nbar"
"foo\nbar"
```

11.4 Einfache Operationen

Arithmetik:
Elixir unterstützt die grundlegenden Operatoren `+`, `-`, `*`, und `/`. Wichtig dabei ist zu beachten, dass `/` immer Float zurück gibt:

```
iex> 2 + 2
4
iex> 2 - 1
1
iex> 2 * 5
10
iex> 10 / 5
2.0
```

Falls du Integerdivision oder den Rest der Division brauchst hat Elixir zwei hilfreiche Funktionen dafür:

```
iex> div(10, 5)
2
iex> rem(10, 3)
1
```

Boolean:

```
iex> -20 || true
-20
iex> false || 42
42

iex> 42 && true
```

```
true
iex> 42 && nil
nil

iex> !42
false
iex> !false
true
```

Es gibt drei weitere Operatoren, deren erstes Argument ein Boolean sein muss:

```
iex> true and 42
42
iex> false or true
true
iex> not false
true
iex> 42 and true
**(ArgumentError) argument error: 42
iex> not 42
**(ArgumentError) argument error
```

Vergleiche:
Elixir kommt mit diversen vergleichenden Operatoren: $==, !=, ===, !==, <=, >=, <$ *und* $>$.

```
iex> 1 > 2
false
iex> 1 != 2
true
iex> 2 == 2
true
iex> 2 <= 3
true
```

Für strikte Vergleiche von Integern und Floats benutze =:

```
iex> 2 == 2.0
true
iex> 2 === 2.0
false
```

Ein wichtiges Feature von Elixir ist, dass jegliche zwei Typen miteinander verglichen werden können. Das ist beispielsweise dann praktisch, wenn man die Typen sortieren möchte. Wir müssen uns nicht an die Sortierreihenfolge erinnern, aber es ist wichtig zu wissen, dass es sie gibt: `number < atom < reference < function < port < pid < tuple < map < list < bitstring`. Das führt mitunter zu interessanten, aber gültigen, Vergleichen, welche es so in anderen Sprachen nicht gibt:

```
iex> :hello > 999
true
iex> {:hello, :world} > [1, 2, 3]
false
```

String Interpolation:
Falls Ruby bereits bekannt ist, wird String Interpolation in Elixir bekannt vorkommen:

```
iex> name = "Sean"
iex> "Hello #{name}"
"Hello Sean"
```

String-Verkettung:
String-Verkettung benutzt den <> Operator:

```
iex> name = "Sean"
iex> "Hello " <> name
"Hello Sean"
```

11.5 Listen

Eine weitere häufige Eigenschaft einer funktionalen Programmiersprache ist, dass viele Operationen an Listen und anderen Sammlungen durchgeführt werden. Listen sind einfache Ansammlungen von Werten, die verschiedene Werte beinhalten dürfen.

```
iex> [3.14, :pie, "Apple"]
[3.14, :pie, "Apple"]
```

Elixir implementiert Listen als verkettete Listen. Das bedeutet der Zugriff auf die Listenlänge ist eine O(n) Operation. Aus diesem Grund ist ist meist schneller ein Element vorne hinzuzufügen als anzuhängen:

```
iex> list = [3.14, :pie, "Apple"]
[3.14, :pie, "Apple"]
```

```
iex> ["  " | list]
["  ", 3.14, :pie, "Apple"]
iex> list ++ ["Cherry"]
[3.14, :pie, "Apple", "Cherry"]
```

Subtraktion:
Unterstützung von Subtraktion ist durch den –/2 Operator gegeben; es ist gefahrlos einen fehlenden Wert zu subtrahieren:

```
iex> ["foo", :bar, 42] -- [42, "bar"]
["foo", :bar]
```

Head/Tail:
Wenn man Listen benutzt ist es üblich mit den Elementen head und tail einer Liste zu arbeiten. head ist das erste Element einer Liste, während tail alle anderen verbleibenden Elemente der Liste darstellt. Elixir bietet zwei hilfreiche Methoden, hd und tl, um mit diesen Teilen zu arbeiten:

```
iex> hd [3.14, :pie, "Apple"]
3.14
iex> tl [3.14, :pie, "Apple"]
[:pie, "Apple"]
```

Zusätzlich zu den bereits erwähnten Funktionen kannst du Pattern Matching und den Cons-Operator | dazu benutzen eine Liste in head und tail zu teilen; wir werden in späteren Kapiteln mehr über dieses Pattern lernen:

```
iex> [head | tail] = [3.14, :pie, "Apple"]
[3.14, :pie, "Apple"]
iex> head
3.14
iex> tail
[:pie, "Apple"]
```

11.6 Tuple

Tupel sind Listen ähnlich, jedoch in angrenzenden Speicheradressen im Speicher abgelegt. Das macht Zugriffe auf ihre Länge schnell, aber Veränderungen kostspielig. Das neue Tupel muss komplett in den Speicher kopiert werden. Sie werden mit geschweiften Klammern geschrieben:

```
iex> {3.14, :pie, "Apple"}
{3.14, :pie, "Apple"}
```

Es ist für Tupel üblich, dass sie als Mechanismus genutzt werden, um zusätzliche Informationen aus einer Funktion zu ziehen. Die Nützlichkeit davon wird offensichtlicher wenn wir Pattern Matching behandeln:

```
iex> File.read("path/to/existing/file")
{:ok, "... contents ..."}
iex> File.read("path/to/unknown/file")
{:error, :enoent}
```

11.7 Maps

In Elixir sind Maps der Weg um einen Key-Value Store zu implementieren. Im Gegensatz zu Keyword Listen erlauben Maps jeden Typ als Schlüssel und sind nicht sortiert. Listen werden mit der %{} Syntax definiert:

```
iex> map = %{:foo => "bar", "hello" => :world}
%{:foo => "bar", "hello" => :world}
iex> map[:foo]
"bar"
iex> map["hello"]
:world
```

Falls ein Duplikat einer Map hinzugefügt wird, wird der neu gesetzte Wert den vorherigen ersetzen:

```
iex> %{:foo => "bar", :foo => "hello world"}
%{foo: "hello world"}
```

Wie wir in der Ausgabe oben erkennen, gibt es eine spezielle Syntax für Maps, die nur aus Atom-Schlüsseln besteht:

```
iex> %{foo: "bar", hello: "world"}
%{foo: "bar", hello: "world"}

iex> %{foo: "bar", hello: "world"} == %{:foo => "bar", :hello => "world"}
true
```

11.8 Funktionen

In Elixir und vielen anderen funktionalen Sprachen sind Funktionen "Bürger erster Klasse". Wir werden mehr über die Typen von Funktionen in Elixir lernen, was sie unterscheidet und wie man sie benutzt.

Wir können Funktionen mit Namen definieren, so dass wir später einfacher auf sie zugreifen können. Benannte Funktionen werden innerhalb eines Moduls mit dem keyword def definiert. Wir werden mehr über Module in der nächsten Lektion lernen, momentan reicht es, wenn wir uns auf benannte Funktionen allein konzentrieren. Funktionen, die innerhalb eines Moduls definiert wurden, sind auch für andere Module nutzbar. Das ist ein besonders nützlicher Baustein in Elixir:

```
defmodule Greeter do
        def hello(name) do
                "Hello, " <> name
        end
end

iex > Greeter.hello("Sean")
"Hello, Sean"
```

Wir haben früher erwähnt, dass Funktionen benannt werden mit der Kombination aus dem Namen und der `arity` (Der Anzahl Argumente). Das bedeutet du kannst Folgendes tun:

```
defmodule Greeter2 do
        def hello(), do: "Hello, anonymous person!"    # hello/0
        def hello(name), do: "Hello, " <> name         # hello/1
        def hello(name1, name2), do: "Hello, #{name1} and #{name2}" # hell
end

iex> Greeter2.hello()
"Hello, anonymous person!"
iex> Greeter2.hello("Fred")
"Hello, Fred"
iex> Greeter2.hello("Fred", "Jane")
"Hello, Fred and Jane"
```

11.9 Weitere Funktionen

Da die Sprache alleine bereits unglaublich gross ist, werden in dieser Einführung nur noch einige besonderheiten angesprochen, die `Elixir` von anderen Sprachen unterscheidet. Allerdings sind die oben genannten Operationen und Datentypen bereits unglaublich start, vor allem wenn man die mitgelieferten

Funktionen dazu nimmt. Sollten noch Unklarheiten zu den oben angesprochenen Datentypen und Operationen herrschen, wird der nächste Abschnitt besonders hilfreich sein.

11.10 Dokumentation

`Elixir` legt grossen Wert auf Dokumentation. Es ist nicht nur möglich sondern sogar der Standard, dass Dokumentation direkt zum Code in die gleiche Datei geschrieben wird. Dafür wird der folgende Syntax verwendet:

```
defmodule Greeter do
        @moduledoc """
        Bietet eine `hello/1` Funktion, um Menschen zu begr  en
        """

        def hello(name) do
                "Hello, " <> name
        end
end
```

Wenn der Code dann fertig ist, kann man ihn mit `hex` veröffentlichen. Dadurch wird es dann allen möglich, die Dokumentation direkt im Browser zu lesen. Auch die Sprache selbst hat ihre Funktionen und Module so Dokumentiert. Als Beispiel gibt es die Dokumentation zu `Maps` unter diesem Link. Meist reicht eine Google Suche nach *Elixir + Maps* um diese Seite zu finden. Da der Source-Code der Sprache öffentlich zugänglich ist, ist es möglich in der Dokumentation einer Funktion direkt zur Definition im Code zu gehen. Im Falle von unserem `Maps` beispiel kann man auf den Link neben einer der Funktionen drücken und kommt direkt auf das `Github-Repository` der Sprache.

11.11 Pattern Matching

Pattern matching ist ein mächtiger Teil Elixirs. Es erlaubt uns einfache Werte, Datenstrukturen und sogar Funktionen zu matchen. Mit korrekter verwendung ist es möglich, viel weniger Code zu schreiben, sowie den geschriebenen Code einiges übersichtlicher zu halten.
Bereit für einen Hirnverdreher? In Elixir ist der =-Operator eigentlich ein match-Operator. Durch diesen können wir Werte zuweisen und matchen:

```
iex> x = 1
```

```
1
iex> 1 = x
1
iex> 2 = x
** (MatchError) no match of right hand side value: 1
```

Lass uns das mit ein paar der collections probieren, die wir kennen:

```
# Lists
iex> list = [1, 2, 3]
iex> [1, 2, 3] = list
[1, 2, 3]
iex> [] = list
** (MatchError) no match of right hand side value: [1, 2, 3]

iex> [1 | tail] = list
[1, 2, 3]
iex> tail
[2, 3]
iex> [2|_] = list
** (MatchError) no match of right hand side value: [1, 2, 3]

# Tupel
iex> {:ok, value} = {:ok, "Successful!"}
{:ok, "Successful!"}
iex> value
"Successful!"
iex> {:ok, value} = {:error}
** (MatchError) no match of right hand side value: {:error}
```

Diese Funktion wird sowohl von der Sprache, als auch von Libraries sehr häufig verwendet. Oftmals sieht man das Folgende:

```
iex> {:ok, file} = File.read("file.txt")
{:ok, "content"}
```

Sollte nun die Datei nicht vorhanden sein, oder es einen anderen Fehler geben, würde die Funktion nicht `{:ok, content}` zurückgeben, sondern {:error, error}. Dabei sind die Atoms an erster stelle nie variablen, sondern fixe Werte, die miteinander verglichen werden können.

11.12 Pipe Operator

Der pipe-Operator |> gibt das Resultat des vorherigen Ausdrucks als ersten Parameter an den neuen Ausdrucks weiter.
Programmieren kann chaotisch sein. Tatsächlich so chaotisch, dass Funktionsaufrufe so eingebettet sind, dass es schwierig wird den Code zu lesen. Nimm zum Beispiel diese verschachtelten Funktionen:

```
iex> foo(bar(baz(new_function(other_function()))))
```

Hier übergeben wir den Wert `other_function/0` an `new_function/1` und `new_function/1` an `baz/1`, `baz/1` an `bar/1` und abschließend das Ergebnis von `bar/1` an `foo/1`. Elixir wählt einen pragmatischen Ansatz, um dieses syntaktische Chaos zu beseitigen, indem es uns den pipe-Operator an die Hand gibt. Der pipe-Operator, der `|>` aussieht nimmt das Ergebnis eines Ausdrucks und gibt es weiter. Lass uns nochmal einen Blick auf das Code Beispiel von oben werfen, diesmal jedoch mit dem pipe-Operator umgeschrieben.

```
iex> other_function() |> new_function() |> baz() |> bar() |> foo()
```

Die pipe nimmt das Ergebnis von links und gibt es an die rechte Seite weiter. Falls du mit Unix vertraut bist, kennst du dieses Verhalten von der Shell und `|`. Beispiel:
Text in Wörter trennen und alle Zeichen in Großbuchstaben umwandeln:

```
iex> "Elixir rocks" |> String.upcase() |> String.split()
["ELIXIR", "ROCKS"]
```

Die obige Einführung und viele weitere Funktionen der Sprache sind auf elixirschool.com[19] erklärt. Dort gibt es eine sehr gute Einführung in die Sprache. Einzelne Funktionen sind auf hexdocs.pm erklärt.

11.13 Module

Es wurde bereits erwähnt, dass `Elixir` eine funktionale Programmiersprache ist. Da die organisatorischen Aspekte von `OOP` eine der grössten Vorteile sind, wurde auch in `Elixir` ein System implementiert, um Funktionen besser zu gruppieren. Ein Modul ist ähnlich wie ein `C#` Namespace. Ein Modul besteht aus einer Sammlung von Funktionen, sowie bis zu einem `struct`, welches mit dem Syntax `%Module.Name{}` erreicht werden kann. Da die Funktionen sowieso keinen Zustand haben kann, spielt es grundsätzlich keine Rolle, in welchem Modul sie sich befindet. Es gibt allerdings einige Regeln

zu Rechten und Zugangsregeln, Details dazu sind auf elixirschool.com[19] im Kapitel Funktionen zu finden.

Kapitel 12

Concurrency

Nun haben wir also den Syntax und einige Funktionen der Sprache kennen gelernt, aber das alleine genügt noch nicht, um ein effizientes *distributed system* zu bauen.
`Elixir` verwendet die gleiche Concurrency Ideologie wie `Erlang`. Wie wir bereits in einem vorherigen Kapitel angeschaut haben funktionieren Anläufe wie `Shared-Memory` nicht gut, oder sind sehr unsicher. Daher baut auch das `Erlang` Model auf `Message passing` auf.
Als erstes müssen die unterschiedlichen Prozesstypen definiert werden:

12.1 Threads

Threads, auch *Real-Threads* genannt, repräsentieren die tatsächlichen Kerne, die auf Hardware Ebene verfügbar sind. Ein Computer mit 4 Kernen kann also maximal 4 Threads offene haben. Dabei wird meist viel Speicher benötigt, um einen solchen Thread zu initialisieren. Da die Anzahl der Threads stark an die Hardware gebunden ist, gibt es ebenfalls grosse Kompatibilitätsprobleme. Aber da die *Threads* direkt zur Hardware gehören, ist die Leistung sehr gross und es kann mehr aus der Hardware heraus geholt werden.
Als gutes Beispiel kann man ein Video-Spiel nehmen. Zumindest neuere Titel laufen meist auf mindestens `4` Kernen und haben dedizierte Prozesse für beispielsweise die Berechnungen für die Physik. Dadurch ist es möglich, die beste Leistung zu erzielen.

12.2 Green-Threads

Während *Real-Threads* sehr teuer zu erstellen sind und deren Anzahl nur begrenzt ist, funktionieren *Green-Threads* etwas verschieden. So ist es möglich eine beinahe unbegrenzte Anzahl *Green-Threads* zu starten. Jeder einzelne *Green-Thread* ist dabei sehr klein und schnell zu starten. Auch benötigt das starten eines *Green-Threads* nur wenige Kilobytes. Daher ist es möglich, innerhalb von kurzer Zeit tausende von diesen *Green-Threads* auf einer einzelnen Maschine zu starten. Allerdings muss auch erkannt werden, dass diese Prozesse weniger Leistung haben, da sie auf einer Abstrahierung der tatsächlichen *Threads* basieren. Daher laufen diese *Threads* nicht tatsächlich Gleichzeitig. Auch wenn man tausende von Prozesse gleichzeitig am Laufen haben kann, wird lediglich zwischen ihnen hin und her gewechselt, wodurch die Leistung natürlich abnimmt. Auch muss erwähnt werden dass für solche Threads eine virtual machine oder etwas ähnliches vorhanden sein muss, allerdings finden wir diese Tools in Sprachen wie `Golang` oder `Erlang`.

12.3 Concurrency

`Erlang` wurde ursprünglich für `Telefonie-Systeme` gebaut, welche eine vielzahl von kleinen Prozessen (Anrufe) gleichzeitig verarbeiten müssen. Daher war es nur logisch, dass *Green-Threads* als Basis für die Sprache gewählt wurden.
In `Erlang`, beziehungsweise `Elixir`, ist es unglaublich einfach, eine Funktion in ihrem eigenen Prozess auszuführen. Nehmen wir als einfaches Beispiel eine Funktion, die zwei Zahlen addiert:

```
defmodule Example do
        def add(a, b) do
                IO.puts(a + b)
        end
end
```

Diese Funktion können wir nun *normal* Ausführen, also nicht in ihrem eigenen Prozess.

```
iex> Example.add(42, 88)
130
:ok
```

Wenn wir nun annehmen, dass die Berechnung innerhalb der Funktion komplizierter sei, oder lange benötigen könnte, macht es Sinn, diese in einem

Prozess zu starten. Dafür müss en wir lediglich eine einzige Änderung im Aufruf der Funktion durchführen.

```
iex> spawn(Example, :add, [42, 88])
130
#PID<0.46.24>
```

Mit diesem Befehl sehen wir nun auch ein `PID` (process identifier). Jeder Prozess bekommt einen solchen per Zufall zugewiesen. Wenn wir dann später mit diesem Prozess kommunizieren möchten, verwenden wir dazu das `PID`. Damit erklären wir auch direkt, wie wir *message passing* in `Elixir` implementieren.

12.4 Message passing

In einem vorherigen Kapitel haben wir bereits die Probleme von *shared-memory* angeschaut. Dieser Anlauf ist weder sicher noch skalierbar. Als alternative wird meist `message passing` verwendet. Dabei können die einzelnen Prozesse Nachrichten und Daten zueinander schicken. Die Daten werden immer selbst geschickt, anstatt einfach Pointers oder Referenzen zu versenden. Daher besitzt jeder Prozess alle seiner Daten selbst und man muss sich keine Sorgen um andere Prozesse und ihre Zustände machen, da jeder Prozess individuell agieren kann.
Schauen wir nun an wie man diese Funktion in `Elixir` implementieren kann. Wieder mit nur sehr wenigen Zeilen Code zu erledigen. Als erstes müssen wir natürlich wieder einen Prozess starten. Dabei ist es wichtig anzumerken, dass unser Vorheriges Beispiel nicht geeignet ist, da der Code lediglich eine Addition durchgeführt hat, wonach sich der Prozess beendet hat. Das ist natürlich unpraktisch, da unser Prozess gestartet bleiben muss, um eine Nachricht zu erhalten.
Nun starten wir einen Prozess, mit dem neuen `receive` Syntax. Man kann sich `receive` wie einen `while(true)` Loop vorstellen, der auf neue Nachrichten wartet.

```
defmodule Example do
        def listen do
                receive do
                        {:ok, "hello"} -> IO.puts("World")
                end

                listen
```

```
        end
end
```

Wir können den Prozess wie gewohnt starten, womit wir dann auch ein `PID` für den Prozess erhalten.

```
iex> pid = spawn(Example, :listen, [])
#PID<0.131.84>
```

In einem nächsten Schritt können wir nun eine Nachricht an den Prozess senden, wofür wir das `send` Schlüsselwort verwenden können. Da wir in der Funktion den Term `{:ok, "hello"}` erwarten, müssen wir auch diesen an die Funktion senden.

```
iex> send pid, {:ok, "hello"}
World
{:ok, "hello"}
```

Zurück erhalten wir dann `:ok` sowie den Rückgabewert der Funktion.
Damit haben wir nun die zwei wichtigsten Komponenten für ein *distributed system*. Die restlichen Funktionen können wir von Hand auf diesen beiden Aufbauen.
Zum Glück aber sind viele Probleme bereits gelöst worden, wodurch wir einfach bereits vorhandene Funktionen benutzen können. Diese finden wir in `OTP`.

Kapitel 13

OTP

In den vorherigen Kapiteln haben wir `Erlang` und `Elixir` angesprochen. Doch was ist `OTP`? Vereinfacht kann man sagen, dass `OTP` eine Sammlung von Funktionen und Verhalten für das Entwickeln von concurrent und fehlertoleranten Programmen für `Erlang` und `Elixir` ist.

13.1 GenServer

Auch wenn der Syntax für das Nachrichten an Prozesse schreiben relativ einfach ist, so gibt es doch einige Probleme damit:

- Wir müssen in jeder Funktion, für die wir diese Funktionalität erlauben möchten, den `receive` loop implementieren.
- Wir haben keine Kontrolle darüber, ob `send` synchron oder asynchron ablaufen soll.
- Es existiert kein Einheitliches System um Zustände mit Prozessen zu verwalten.

Mit einem `GenServer` haben wir Zugang zu einer Abstrahierung, die es uns erlaubt, die Probleme mit einem einheitlichen System anzugehen.
Um `GenServer` und viele andere Aspekte von `OTP` zu verstehen, müssen wir als erstes kurz Verhalten oder auch `Behaviours` anschauen:
Wenn wir ein Modul haben, können wir ein `Behaviour` implementieren. Dann müssen wir gewisse Funktionen implementieren, die dann von dem importieren Code aufgerufen werden.

Für einen `GenServer` können wir einfach das gleich genannte `Behaviour` importieren:

```
defmodule Some.Module do
        use GenServer
end
```

Auch um `GenServer` zu erklären wollen wir die offizielle Dokumentation auf elixirschool.com[19] verwenden:
Auf dem untersten Level ist ein `GenServer` eine Schleife, welche einen Request pro Iteration handhabt, indem sie einen aktualisierten Status herum reicht.

Um die `GenServer-API` zu demonstrieren, werden wir eine einfache Queue implementieren, die Werte speichert und entgegen nimmt.

Um unseren `GenServer` anzufangen, müssen wir ihn starten und die Initialisierung regeln. In den meisten Fällen wollen wir Prozesse miteinander verbinden, so dass wir `GenServer.start_link/3` benutzen. Wir übergeben das GenServer-Modul, das wir starten, initiale Argumente und ein Set an GenServer-Optionen. Die Argumente werden an GenServer.init/1 übergeben, was wiederum den initialen Status durch den Rückgabewert setzt. In unserem Beispiel sind die Argumente der initiale Status:

```
defmodule SimpleQueue do
        use GenServer

        @doc """
        Start our queue and link it.  This is a helper function
        """
        def start_link(state \\ []) do
                GenServer.start_link(__MODULE__, state, name: __MODULE__)
        end

        @doc """
        GenServer.init/1 callback
        """
        def init(state), do: {:ok, state}
end
```

Oft ist es notwendig mit unserem `GenServer` in einer synchronen Art und Weise zu interagieren, etwa eine Funktion aufrufen und auf das Ergebnis warten. Um synchrone Requests zu verwalten müssen wir den GenServer.handle$_{\text{call}}$/3-Callback benutzen, welcher benötigt: Den Request, den PID des Aufrufers

und den vorhandenen Status; es wird davon ausgegangen, dass er ein Tupel zurückgibt: {:reply, response, state}.
Mit pattern matching können wir Callbacks für viele verschiedene Requests und Zustände definieren. Eine komplette Liste akzeptierter Rückgabewerte findet sich in der GenServer.handle$_{\text{call}}$/3-Dokumentation.
Um synchrone Requests zu demonstrieren lass uns die Möglichkeit einbauen, unsere aktuelle Queue anzusehen und einen Wert rauszunehmen:

```
defmodule SimpleQueue do
        use GenServer

        ### GenServer API

        @doc """
        GenServer.init/1 callback
        """
        def init(state), do: {:ok, state}

        @doc """
        GenServer.handle_call/3 callback
        """
        def handle_call(:dequeue, _from, [value | state]) do
                {:reply, value, state}
        end

        def handle_call(:dequeue, _from, []), do: {:reply, nil, []}

        def handle_call(:queue, _from, state), do: {:reply, state, state}

        ### Client API / Helper functions

        def start_link(state \\ []) do
                GenServer.start_link(__MODULE__, state, name: __MODULE__)
        end

        def queue, do: GenServer.call(__MODULE__, :queue)
        def dequeue, do: GenServer.call(__MODULE__, :dequeue)
end
```

Nun können wir die SimpleQueue startet und die neue dequeue-Funktionalität testen:

```
iex> SimpleQueue.start_link([1, 2, 3])
{:ok, #PID<0.90.0>}
iex> SimpleQueue.dequeue
1
iex> SimpleQueue.dequeue
2
iex> SimpleQueue.queue
[3]
```

Asynchrone Requests werden durch den $\text{handle}_{\text{cast}}/2$-Callback behandelt. Dieser arbeitet ähnlich wie $\text{handle}_{\text{call}}/3$, bekommt jedoch keinen Aufrufer übergeben und es wird nicht davon ausgegangen, dass er eine Rückgabe hat. Wir werden unsere enqueue-Funktionalität asynchron implementieren. Die Queue wird aktualisiert, jedoch blockiert der Aufruf nicht unsere aktuelle Ausführung:

```
defmodule SimpleQueue do
        use GenServer

        ### GenServer API

        @doc """
        GenServer.init/1 callback
        """
        def init(state), do: {:ok, state}

        @doc """
        GenServer.handle_call/3 callback
        """
        def handle_call(:dequeue, _from, [value | state]) do
                {:reply, value, state}
        end

        def handle_call(:dequeue, _from, []), do: {:reply, nil, []}

        def handle_call(:queue, _from, state), do: {:reply, state, state}

        @doc """
        GenServer.handle_cast/2 callback
        """
        def handle_cast({:enqueue, value}, state) do
```

```
            {:noreply, state ++ [value]}
        end

        ### Client API / Helper functions

        def start_link(state \\ []) do
            GenServer.start_link(__MODULE__, state, name: __MODULE__)
        end

        def queue, do: GenServer.call(__MODULE__, :queue)
        def enqueue(value), do: GenServer.cast(__MODULE__, {:enqueue, value
        def dequeue, do: GenServer.call(__MODULE__, :dequeue)
end
```

Nun können wir die neue Funktionalität ausprobieren:

```
iex> SimpleQueue.start_link([1, 2, 3])
{:ok, #PID<0.100.0>}
iex> SimpleQueue.queue
[1, 2, 3]
iex> SimpleQueue.enqueue(20)
:ok
iex> SimpleQueue.queue
[1, 2, 3, 20]
```

Als weiteres Beispiel lässt sich auch noch `gen_demo`, ein Ausschnitt aus einem anderen Software-Projekt, empfehlen. Auch wenn der Kontext etwas unklar sein kann sind noch einige relevante Mechanismen im Code die später auch noch sehr wichtig werden. Da aber der Code über viele Dateien verteilt ist hat er keinen Platz in dieser Arbeit gefunden. Da der Code später noch wichtig wird ist es also wichtig das der dortige Code genau so gut verstanden ist, wie die obigen Beispiele. Der Code ist unter gendemo.jeykey.net zu finden.

13.2 Supervisor

Supervisors sind besondere Prozesse mit einem Zweck: andere Prozesse zu überwachen. Diese Supervisors erlauben uns fehlertolerante Anwendungen zu erstellen, indem sie Kindprozesse automatisch neu starten, falls diese versagen.

Die Magie von Supervisors liegt in der Supervisor.start$_{\text{link}}$/2-Funktion. Zusätzlich zum Starten unserer Supervisors und Kinder erlaubt sie uns die

Strategie zu bestimmen, mit der unser Supervisor Kindprozesse verwaltet.

Um ein neues Projekt zu erstellen können wir den Befehl `mix new simple_queue --sup` verwenden. Damit erhalten wir auch direkt einen Supervisor-Baum (Also die grundlegende Code Struktur). Der Code für das Modul SimpleQueue sollte in `lib/simple_queue.ex` liegen und der Supervisor Code, den wir hinzufügen werden, soll in lib/simple$_{\text{queue}}$/application.ex liegen.
Kinder werden werden in einer Liste definiert, entweder eine Liste von Modulnamen:

```
import Supervisor.Spec

children = [
        worker(SimpleQueue, [], name: SimpleQueue)
]

{:ok, pid} = Supervisor.start_link(children, strategy: :one_for_one)
```

oder eine Liste von Tupeln, falls man Konfigurations-Optionen mitgeben will:

```
defmodule SimpleQueue.Application do
        use Application

        def start(_type, _args) do
                children = [
                        {SimpleQueue, [1, 2, 3]}
                ]

                opts = [strategy: :one_for_one, name: SimpleQueue.Supervis
                Supervisor.start_link(children, opts)
        end
end
```

Wenn wir `iex -S mix` ausführen, sehen wir, dass unsere SimpleQueue automatisch gestartet wird:

```
iex> SimpleQueue.queue
[1, 2, 3]
```

Falls unser SimpleQueue Prozess abstürzt oder sonstwie terminiert, würde unser Supervisor ihn automatisch neu starten, als ob nichts gewesen wäre. Es gibt zurzeit drei verschiedene Strategien zum Neustart, welche Supervisors benutzen können:

- :one_for_one - Startet nur den abgestürzten Kindprozess neu.
- :one_for_all - Startet alle Kindprozesse neu im Falle eines Fehlers.
- :rest_for_one - Startet den abgestürzten Prozess und alle nach ihm gestarteten Prozesse neu.

Nachdem ein Supervisor gestartet ist, muss er wissen, wie er seine Kinder starten/stoppen/neustarten soll. Jedes Kind-Modul sollte eine $\text{child}_{\text{spec}}/1$ Funktion haben, die dieses Verhalten definiert. Die Macros `use GenServer`, `use Supervisor` und `use Agent` definieren diese Methode automatisch für uns (SimpleQueue hat `use Genserver`, also brauchen wir dieses Modul nicht anzupassen), aber wenn du sie selbst definieren musst, sollte $\text{child}_{\text{spec}}/1$ eine Map von Optionen zurückgeben:

```
def child_spec(opts) do
        %{
                id: SimpleQueue,
                start: {__MODULE__, :start_link, [opts]},
                shutdown: 5_000,
                restart: :permanent,
                type: :worker
        }
end
```

- `id`: Notwendiger Key. Vom Supervisor benutzt um die die Kind-Spezifikation zu identifizieren.
- `start`: Notwendiger Key. Das Modul, die Funktion und Argumente, die beim Start vom Supervisor aufgerufen werden sollen.
- `shutdown`: Optionaler Key. Definiert das Verhalten des Kindes während des Beendens Die Optionen sind:
  - `:brutal_kill`: Kind wird sofort gestoppt
  - ein positiver Integer - Zeit in Millisekunden, die der Supervisor warten wird, bevor er den Kind-Prozess killt. Wenn der Prozess vom Typ `:worker` ist, ist dieser Wert standardmäßig `5000`.
  - `:infinity`: Der Supervisor wird unendlich lange warten, bevor er den Prozess killt. Standardwert für den Prozesstyp `:supervisor`. Nicht empfohlen für den Typ `:worker`.

- `restart`: Optionaler Key. Es gibt mehrere Herangehensweisen, um abstürzende Kindprozesse zu verwalten:
  - `:permanent`: Kind wird immer neugestartet. Standard für alle Prozesse.
  - `:temporary`: Kindprozess wird niemals neugestartet.
  - `:transient`: Kindprozess wird nur neu gestartet, falls er abnorm terminiert.
- `type`: Optionaler Key. Prozesse können entweder :worker oder :supervisor sein. Standardwert ist `:worker`.

13.3 DynamicSupervisor

Supervisors starten normalerweise mit einer Liste von Kindern, die sie starten, wenn die App startet. Manchmal jedoch sind die beaufsichtigen Kinder beim App-Start nicht bekannt (zum Beispiel könnten wir eine Web-App haben, die einen neuen Prozess startet, wenn sich ein Nutzer mit unserer Webseite verbindet.) Für diese Fälle brauchen wir eine Supervisor der Kinder bei Bedarf starten kann. In diesem Fall benutzen wir den `DynamicSupervisor`. Da wir keine Kinder spezifizieren, müssen wir nur die Laufzeit-Optionen für den Supervisor bestimmen. Der DynamicSupervisor unterstützt nur die Supervison-Strategie `:one_for_one`:

```
options = [
        name: SimpleQueue.Supervisor,
        strategy: :one_for_one
]

DynamicSupervisor.start_link(options)
```

Um eine neue SimpleQueue dynamisch zu starten, nutzen wir dann die Funktion $\text{start}_{\text{child}}/2$, welche einen Supervisor und eine Kind-Spezifikation als Argumente nimmt (SimpleQueue nutzt use GenServer, die Kind-Spezifikation ist daher schon definiert):

```
{:ok, pid} = DynamicSupervisor.start_child(
SimpleQueue.Supervisor,
SimpleQueue
)
```

13.4 Name registration

Ein Namensregister ermöglicht es Entwicklern, einen oder mehrere Prozesse mit einem bestimmten Schlüssel nachzuschlagen. Wenn die Registrierung `:unique` Schlüssel hat, zeigt ein Schlüssel auf 0 oder 1 Prozesse. Wenn die Registrierung `:duplicate` Schlüssel erlaubt, kann ein einzelner Schlüssel auf eine beliebige Anzahl von Prozessen zeigen. In beiden Fällen könnten verschiedene Schlüssel denselben Prozess identifizieren. Jeder Eintrag in der Registrierung ist dem Prozess zugeordnet, der den Schlüssel registriert hat. Wenn der Prozess abstürzt, werden die diesem Prozess zugeordneten Schlüssel automatisch entfernt. Die Registrierung kann für verschiedene Zwecke verwendet werden, wie z. B. Namenssuche (mit der Option :via), Speichern von Eigenschaften, benutzerdefinierte Dispatching-Regeln oder eine Pubsub-Implementierung. Wir untersuchen einige dieser Anwendungsfälle weiter unten. Die Registry kann auch transparent partitioniert sein, was ein besser skalierbares Verhalten bei der Ausführung von Registrierungen auf hochgradig gleichzeitigen Umgebungen mit Tausenden oder Millionen von Einträgen ermöglicht.

Um eine solche Funktion nun mit unserem `GenServer` zu verwenden, können wir einfach eine `:name` option hinzufügen.

Wir haben drei optionen, um einen `GenServer` zu registrieren:

- Ein beliebiges `Atom`, womit wir die eingebaute `Elixir` Registry <verwenden.
- `{:global, term}` um den `GenServer` mit Hilfe des `:global` modules zu verwenden. Dadurch können wir einen registrieren Prozess auf allen verbundenen Nodes erreichen. Allerdings ist diese Variante nur bedingt skalierbar und eher langsam.
- `{:via, module, term}` um den `GenServer` mit einem eigenen Module zu registrieren. Ein solches Module muss das passende Verhalten implementieren. Dafür könnte man beispielsweise das `:pg2` Modul der `Erlang` Sprache verwenden.

Eine einfache lokale Registrierung könnte wie folgt aussehen:

```
GenServer.start_link(__MODULE__, state, name: :some_name)
```

13.5 Clustering

Tatsächlich macht es uns Elixir und Erlang unglaublich einfach, mehrere Nodes zusammenzuschliessen. Wenn man einfach alle Nodes verbindet, funktioniert das gesamte System als eine einzige Instanz. Jede Node kann also Requests verarbeiten. Tatsächlich haben wir einen Teil von distributed erlang gesehen, als wir PIDs angeschaut haben. Ein PID besteht immer aus drei Zahlen, wobei die erste die Node symbolisiert.
Wir können also von jeder Erlang Node eine Nachricht zu einer beliebigen anderen schicken (Einer der grössten Vorteile eines fully connected mesh).
Die verschiedenen Varianten an *distributed systems* haben wir bereits angeschaut, daher reicht es hier zu erwähnen, das Erlang und OTP ein fully connected mesh nutzen. Dies hat natürlich einige Probleme. Die meisten davon sind allerdings nicht besonderheiten der Sprache, daher müssen diese hier nicht nochmals erwähnt werden.
Was allerdings dieses Ökosystem besonders gut geeignet für solche Systeme macht, ist der Fakt dass so gut wie keine andere Sprache ähnliche Funktionen überhaupt unterstützt. Auch wenn das Senden von Nachrichten zwischen Prozessen mit HTTP oder etwas ähnlichen nachgebaut werden kann, gibt es sonst nirgends ein einheitliches System, Zustände und Daten zu verwalten.

13.6 ETS

Während die meisten Programmiersprache einfache Zugangsmöglichkeiten zu Datenbanken haben, so kommt OTP mit drei eigenen Datenbanken, welche man direkt in der Sprache verwenden kann.
Erlang Term Storage, allgemein als ETS bezeichnet, ist eine leistungsstarke Speicher-Engine, die in OTP integriert und für die Verwendung in Elixir verfügbar ist.
Für diesen Abschnitt werden einige Grundkenntnisse über Datenbanken und die Rolle die diese in *distributed systems* spielen angenommen.
In vielerlei hinsicht unterscheidet sich ETS von anderen Datenbanken, wie PostgreSQL oder Riak. Da PostgreSQL ein eigenes Ökosystem besitzt, müssen gespeicherte Werte immer von PostgreSQL zu Typen einer Sprache umgewandelt werden. Dadurch entstehen natürlich ineffizienzen. ETS ist stattdessen direkt in die Sprache integriert und wie der Name verrät speichert man in ETS Erlang *terms* direkt. ETS ist vergleichbar zu Riak, da es sich ebenfalls um einen KVS (Key - Value - Store) handelt. Man kann also einen

beliebigen Schlüssel als Erlang-Typ mit beliebigen anderen Datentypen als Werte in dieser Datenbank speichern. Auch ist es wichtig zu verstehen, dass es keinerlei Schemas oder Migrationen gibt. ETS selbst läuft in einem normalen OTP Prozess. Wenn dieser Abstürzt, können Prozess und Table einfach neu gestartet werden. Aber ETS selbst ist nicht Teil von *distributed Erlang*. Ein ETS Table kann also nur auf der eigenen Node erreicht werden.
Natürlich kann man den Prozess, auf dem ETS läuft mit :global oder etwas ähnlichen registrieren und dann einfach Nachrichten senden, aber wir werden später noch sehen, dass es dafür eine eigene Datenbank gibt.
Eine wichtige Feststellung ist ebenfalls, dass ETS nur in-memory speichert. Dies bedeutet, dass die Daten nicht auf die Festplatte geschrieben werden und theoretisch verloren gehen können, sollte der Prozess abstrützen.

13.7 DETS

Disk Erlang Term Storage ist sehr ähnlich zu ETS mit dem grossen Unterschied, dass die Werte tatsächlich auf die Festplatte geschrieben werden. Dadurch ist DETS natürlich etwas langsamer. Es kommen noch weitere komplikationen hinzu, wenn die Datenmenge grösser als der verfügbare Arbeitsspeicher ist. Aber da die Sprache und diese Datenbanken schon sehr lange in verwendung sind, gibt es haufenweise Programme die sich um solche Fälle kümmern.

13.8 Mnesia

Während sowohl ETS als auch DETS nur für eine Node funktionieren, ist Mnesia die Erlang Lösung für *distributed systems*. Dabei baut Mnesia auf ETS und DETS auf, wobei diese lokal verwendet werden. Dazu kommen Mechaniken für die Kommunikation zwischen den ETS Tables.
Die API ist dabei ähnlich wie für ETS und DETS. Mnesia bietet viele Konfigurationsmöglichkeiten und Optionen für die Nutzer, wodurch es vielseitig einsetzbar ist. Mnesia ist sowohl mit als auch ohne Disk Storage verfügbar. Leider bieter Mnesia etwas weniger flexibilität als andere Komponenten des Ökosystems. Beispielsweise das nachträgliche Hinzufügen von Nodes ist oftmals problematisch.

13.9 Probleme

Es ist wichtig zu verstehen, dass `Mnesia` und andere `OTP` Tools auf den ersten Blick magisch wirken können. So als ob sie alle Probleme mit `distributed systems` lösen. Aber tatsächlich sind die meisten nur Abstrahierungen einer sehr einfachen `Message Passing` Mechanik.
`OTP` bietet lediglich einige Funktionen, die es einfacher und schöner machen `distributed systems` schreiben. Da es für `Elixir` und `Erlang` auch haufenweise Libraries und Tools gibt, muss man etwas aufpassen, da viele nicht einfach nur Funktionen zur verfügung stellen und den Programmierer entscheiden lassen wie diese zu verwenden sind, sondern tatsächlich auch Ideologien und Entwicklungsarten durchsetzen. Je nach Kombination dieser Tools kann es zu einem Durcheinander verschiedener dieser Prinzipien kommen.
Trotzdem erlaubt es `Elixir` die Geschwindigkeit der Entwicklung drastisch zu erhöhen ohne Kompromisse im Zusammenhang mit Sicherheit oder Zuverlässigkeit.

Aufgaben:

- Wieso gibt es noch kein *perfektes distributed system* und wann können wir mit einem rechnen?
- Im Text wurde als Beispiel für ein *distributed system* auch soziale Interkationen genannt. Wie passen die Kriterien, Bedingungen und Architekturen auf dieses Beispiel?
- Haben Revolutionen wie *Blockchain* eine Chance populär für solche Systeme zu werden? Wieso, oder wieso nicht?
- Wie unterscheiden sich *distributed systems* wie das Internet und die Systeme die einen Server ausmachen.
- Finde drei Beispiele für jede Kategorie: `AP`, `CP`, `AC`.
- Installiere `Erlang` und `Elixir` auf dem eigenen Laptop, wenn möglich auf mehreren Geräten.
- Mache dich mit der `Eliixr` Shell und den buildtools der Sprache vertraut und spiele etwas mit ihnen herum.
- Schreibe einige Beispielprojekte in `Elixir` wie beispielsweise Fibonacci oder FizzBuzz.
- Schreibe einige Beispielprojekte mit `Elixir` `Erlang` Concurrency.
- Versuche diese Beispielprojekte oder gen_{demo} auf mehreren Geräten oder Instanzen zu starten und diese mit *distributed Erlang* zu verbinden.
- Experimentiere mit `ETS` und erstelle einen Table. Dafür brauchst du wahrscheinlich die offizielle Dokumentation.
- Teste auch diese Version mit *distributed Erlang* und schreibe einen Wert von einer anderen Node.
- Entwickle ein einfaches Spiel oder sinnvolles Programm welches von Concurrency nutzen macht und welches einen praktsichen Nutzen hat (Präsentiere dieses).

Literaturverzeichnis

[1] Wikipedia, *Distributed computing*, https://en.wikipedia.org/wiki/Distributed_compu heruntergeladen am: 17.05.2020.

[2] Riak, *The world's most resilient NoSQL database*, https://riak.com, heruntergeladen am: 26.05.2020.

[3] InfoQ, *A Critique of Resizable Hash Tables: Riak Core Random Slicing*, https://www.infoq.com/articles/dynamo-riak-random-slicing/, heruntergeladen am: 22.05.2020.

[4] Apache Kafka, *A distributed streaming platform*, https://kafka.apache.org/, heruntergeladen am: 22.05.2020.

[5] Wikipedia, *Transatlantic telegraph cable*, https://en.wikipedia.org/wiki/Transatlantic_telegraph_cable, heruntergeladen am: 23.05.2020.

[6] This day in history, *First transatlantic telegraph cable completed*, https://www.history.com/this-day-in-history/first-transatlantic-telegrap heruntergeladen am: 23.05.2020.

[7] InfoQ, *55th Anniversary of Moore's Law*, https://www.infoq.com/news/2020/04/Moores-law-55/, heruntergeladen am: 19.05.2020.

[8] Harald Pöttinger, *Peer-to-Peer-Systeme bilden die Basis der Blockchain*, http://haraldpoettinger.com/peer-to-peer/, heruntergeladen am: 21.05.2020.

[9] Wikipedia, *Fallacies of Distributed Computing*, https://de.wikipedia.org/wiki/Fallacies_of_Distributed_Computing, heruntergeladen am: 19.05.2020.

[10] Wikipedia, *Hash function*, https://en.wikipedia.org/wiki/Hash_function, heruntergeladen am: 20.05.2020.

[11] Wikipedia, *Consistent hashing*, https://en.wikipedia.org/wiki/Consistent_hashing, heruntergeladen am: 20.05.2020.

[12] WebRTC *Real-time communication for the web*, https://webrtc.org/, heruntergeladen am: 26.05.2020.

[13] WebRTC, *TURN server*, https://webrtc.org/getting-started/turn-server, heruntergeladen am: 26.05.2020.

[14] WebRTC Glossary, *SFU (Selective Forwarding Unit)*, https://webrtcglossary.com/sfu/, heruntergeladen am: 26.05.2020.

[15] Ribbon, *What is a WebRTC Gateway?*, https://ribboncommunications.com/company/get-help/glossary/webrtc-gatewa heruntergeladen am: 26.05.2020.

[16] Golang talks, *Concurrency is not Parallelism*, https://talks.golang.org/2012/waza.slide, heruntergeladen am: 21.05.2020.

[17] Dynamo Paper, *Dynamo: Amazon's Highly Available Key-value Store*, https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf, heruntergeladen am: 26.05.2020.

[18] OpenLab, *For a little perspective ...*, https://openlab.citytech.cuny.edu/neh-digitech/2016/02/17/for-a-little-p heruntergeladen am: 26.05.2020.

[19] ElixirSchool, *The premier destination for learning and mastering Elixir*, https://elixirschool.com, heruntergeladen am: 26.05.2020.

[20] YouTube, *Avoiding Microservice Megadisasters*, https://youtu.be/gfh-VCTwMw8, heruntergeladen am: 26.05.2020.

[21] Wikipedia, *Domain Name System*, https://en.wikipedia.org/wiki/Domain_Name_Syst heruntergeladen am: 26.05.2020.

[22] Datafloq, *What Data Do The Five Largest Tech Companies Collect*, https://datafloq.com/read/what-data-do-the-five-largest-tech-companies-c heruntergeladen am: 26.05.2020.

[23] Robert on anything, *Virding's First Rule of Programming*, http://rvirding.blogspot.com/2008/01/virdings-first-rule-of-programming. heruntergeladen am: 26.05.2020.

[24] YouTube, *The Two Generals' Problem*, https://youtu.be/IP-rGJKSZ3s, heruntergeladen am: 27.05.2020.

[25] Apache Cassandra, *Manage massive amounts of data, fast, without losing sleep*, https://cassandra.apache.org/, heruntergeladen am: 27.05.2020.

[26] Ruby Programmiersprache, *DER BESTE FREUND EINES PROGRAMMIERERS*, https://de.wikipedia.org/wiki/Ruby_(Programmiersprache), heruntergeladen am: 29.05.200.