KANTI BADEN

Engine: Orion

Dominik Keller, Jakob Klemm, G3a

Maturaarbeit, Kanti-Baden

Simon Hallström, 9.06.2021

Inhaltsverzeichnis

Egal wie sicher wir uns in unserem digitalen Umfeld fühlen, und egal wie abhängig wir vom modernen Informationsaustausch sind, das kleinste Problem in einem der vielen Produkte und Dienste, die wir täglich für selbstverständlich halten, könnte unsere gesamte Art zu leben grundlegend verändern. Zwar mag es so wirken, als wäre alles in Ordnung, aber in Wahrheit gibt es eine Vielzahl von Katastrophen und Problemen, die uns jeden Moment unvorbereitet treffen könnten. Limitierte IP-Adressen, überlastete Router, die enorme Zentralisierung und unbrauchbare Entwicklungssysteme sind einige der wichtigsten Probleme unserer Zeit.
Aufbauend auf einem verteilten Daten- und Nachrichtensystem bietet `Engine: Orion` verschiedene Mechanismen und Programme für eine neue Art der digitalen Kommunikation und des Informationsaustausches. Es geht dabei nicht darum, die alten Probleme zu beheben oder temporäre Lösungen zu finden. `Engine: Orion` ist eine unabhängige Alternative, ein Neustart, der aus den Fehlern der Vergangenheit lernen kann, um ein robustes, skalierbares, globales Datensystem zu entwickeln, welches nicht nur für Firmen oder Experten zugänglich ist, sondern es allen erlaubt, an diesem Datenaustausch teilzunehmen.

`Engine: Orion` erreicht dies durch ein modulares Zusammenspiel verschiedener Komponenten und der Verwendung von *open-source*, gut getesteten Werkzeugen, Ideen und Prinzipien. `Engine: Orion` erlaubt es Entwicklern innert kürzester Zeit eigene, dezentralisierte Applikationen zu entwickeln und direkt an Nutzer zu liefern, ohne dass die Bequemlichkeit der Nutzer kompromittiert wird.

Aktuell bietet `Engine: Orion` einen ersten Prototypen dieser Vision. Durch ein komplexes Zusammenspiel verschiedener Komponenten und Geräte können Nachrichten einer einfachen `Chat`-Applikation über ein eigenes Verarbeitungssystem verteilt und gespeichert werden. Dabei spielt es keine Rolle, wo im System die eigentliche Verarbeitung stattfindet.

Die aktuelle Demo ist noch weit von der weltverändernden Revolution entfernt, die von Nöten wäre, um die Katastrophe langfristig zu verhindern. Trotzdem ist es ein signifikanter erster Schritt, der das Potential hat, grosse Veränderungen in der Zukunft mit sich zu bringen.

1 Vision

1. Oktober 1964 - UCLA an Stanford: LO

1964 rechnete niemand mit der fundamentalen Änderung unserer Existenz und Lebensweise, die mit dieser einfachen Nachricht in Bewegung gebracht wurde. Eigentlich hätte die erste Nachricht über das `ARPANET` im Jahre 1964 `LOGIN` heissen sollen, doch das Netzwerk stürzte nach nur zwei Buchstaben ab. Ob dies als schlechtes Omen für die Zukunft hätte gewertet werden sollen bleibt eine ungeklärte Frage. Aber das Internet ist hier und es ist so dominant wie noch nie zuvor. Jetzt ist es in der Verantwortung jeder neuen Generation auf diesem Planeten, mit den unglaublichen Möglichkeiten richtig umzugehen und die Vielzahl an bevorstehenden Katastrophen und Gefahren zu navigieren.

Ohne das Internet wäre die Welt wie wir sie kennen nicht möglich. Unsere Arbeit, Kommunikation und unser Entertainment sind nicht einfach nur abhängig von der enormen Interkonnektivität des Internets, ohne sie würden ganze Industrien und Bereiche unser Gesellschaft gar nicht erst existieren. Das Internet hatte einen selbstverstärkenden Effekt auf sein eigenes Wachstum. Der um ein Vielfaches schnellere Datenaustausch und die enorme Interkonnektivität führten dazu, dass jede neue Innovation und jede neue Plattform im Internet noch schneller noch mehr User erreichte und auf immer unvorstellbarere Grössen anwuchs.

Das ist ja grundsätzlich nichts Schlechtes. Das Internet hat eine unvorstellbare Menge an Vermögen, Geschwindigkeit und Bequemlichkeit für uns alle geschaffen und wir haben unsere Gesellschaftsordnung daran ausgerichtet. Aber man muss sich fragen, ob wir manche der Schritte nicht doch überstürzt haben. Im Namen des Wachstums und aus `FOMO` (*Fear Of Missing Out*) wurden Technologien für die Massen zugänglich, die eigentlich nie für solche Grössenordnungen entwickelt wurden. Denn sobald die immer höheren Erwartungen an teils unglaublich fragile Systeme nicht mehr erfüllt werden, kommt es schnell zur Katastrophe. Und durch unsere Abhängigkeit von diesen Systemen steht bei einem solchen Szenario nicht nur der Untergang einiger Produkte oder einzelner Firmen bevor, nein, es könnte zum Kollaps ganzer Länder oder Gesellschaften kommen.

Egal wie sicher und zuverlässig unsere *öffentliche* Infrastruktur auch scheinen mag, es lassen sich doch schnell Risse im System erkennen. Nicht nur

an der Oberfläche, sondern auch im Herzen unseres digitalen Leben gibt es Probleme. Oftmals handelt es sich dabei nicht um *Kleinigkeiten*, *Meinungsverschiedenheiten* oder *Kontroversen*, sondern um physikalische Grenzen, grundlegende Designfehler und das vielseitige Versagen der involvierten Parteien.

In den nächsten Kapiteln sollen einige dieser zentralen Probleme besprochen werden. Dabei soll versucht werden nicht nur die fehlerhaften Implementierungen zu erklären, sondern auch die dadurch entstandenen Probleme in Verbindung mit unseren täglichen Interaktionen und Verwendungen des Internets zu bringen. In einem nächsten Schritt soll dann eine Lösung besprochen werden: ein System, mit welchem sich möglichst viele der grössten Probleme lösen lassen, und welches tatsächlich praktischen Nutzen bietet.

1.1 IP-V4

In der Geschichte der Menschheit haben wir aus vielen verschiedenen Gründen Krieg geführt. Für Wasser, Nahrung, Öl, Frieden oder Freiheit in den Krieg zu ziehen, scheint zu einer fernen Welt zu gehören. Aber auch wenn diese grundlegenden Verlangen gedeckt sind, werden schon bald neue Nöte aufkommen. Während *Daten* oft als Gold des 21. Jahrhunderts bezeichnet werden, gibt es noch eine andere Ressource, deren Vorräte wir immer schneller erschöpfen.

$4'294'967'296$. So viele `IP-V4`-Adressen wird es jemals geben. `IP-V4`-Adressen werden für jedes Gerät benötigt, das im Internet kommunizieren will und dienen zur eindeutigen Identifizierung. Aktuell wird die vierte Version (`V4`) verwendet. In einer Wirtschaft, in der unendliches Wachstum als letzte absolute Wahrheit geblieben ist, kann ein solch hartes Limit verheerende Folgen haben. Besonders wenn die limitierte Ressource so unendlich zentral für unser aller Leben ist, wie nichts anderes. Mit `IP-V6` wird zurzeit eine Alternative angeboten, die solche Limitierungen nicht hat. Aber der Wechsel ist eine freiwillige Entscheidung, für die nicht nur alle Betroffenen bereit sein müssen, sondern für die auch jede einzelne involvierte Komponente diese neue Technologie unterstützen müssen.

1.2 Routing

Freiheit und Unabhängigkeit sind menschlich. Es darf niemals bestraft werden, nach diesen fundamentalen Rechten zu streben. Und doch führt das egoistische Streben nach Freiheit zu Problemen, oftmals allerdings nicht für die nach Freiheit Strebenden.

Genau diese Situation findet man im aktuellen Konflikt um die Grösse von *Address-Abschnitten* vor. Um dieses Problem richtig zu verstehen, muss als erstes die Funktion der *Zentralrouter* und der globalen Netzwerkinfrastruktur erklärt werden:

Jedes Gerät im Internet ist über Kabel oder Funk mit jedem anderen Gerät verbunden. Da das Internet aus einer Vielzahl von Geräten besteht, wäre es unmöglich, diese direkt miteinander zu verbinden. Daher lässt sich das Internet besser als *umgekehrte Baum-Struktur* vorstellen:

- Ganz unten finden sich die Blätter, die Abschlusspunkte der Struktur. Sie stellen die *Endnutzergeräte* dar. Jeder Server, PC und jedes `iPhone`. Hier ist es auch wichtig festzustellen, dass es in dieser Ansicht des Internets keine magische *Cloud* oder ferne Server und Rechenzentren gibt. Aus der Sicht des Netzwerks sind alle Endpunkte gleich, auch wenn manche für Konsumenten als *Server* gelten.

- Die Verzweigungen und Knotenpunkte über den Blättern, dort wo sich Äste aufteilen, stellen *Router* und Switches dar. Hier geht es allerdings nicht um Geräte, die sich in einem persönlichen Setup oder einem normalen Haushalt finden. Mit Switches sind die Knotenpunkte (`POP-Switches`) der Internet-Anbieter gemeint. Diese teilen eingehende Datenströme auf und leiten die richtigen Daten über die richtigen Leitungen.

- Ganz oben findet sich der Stamm. Während ein normaler Baum natürlich nur einen Stamm hat, finden sich in der Infrastruktur des Internets aus Zuverlässigkeitsgründen mehrere. Von diesen `Zentralroutern` gibt es weltweit nur eine Handvoll und sie sind der Grund für das Problem.

Die `Zentralrouter` kümmern sich nicht um einzelne Adressen, sondern um Abschnitte von Adressen, auch `Address Spaces` genannt. An den zentralen Knotenpunkten geht es also nicht um einzelne Server oder Geräte, zu

dem etwas gesendet werden muss, stattdessen wird eher entschieden, ob gewisse Daten beispielsweise von Frankfurt aus nach Ost- oder Westeuropa geschickt werden müssen.

Im Laufe der Jahre wurden die grossen Abschnitte von Adressen aber immer weiter aufgeteilt. Internet-Anbieter und grosse Firmen können diese Abschnitte untereinander verkaufen und aufteilen. Und jede Firma will natürlich ihren eigenen Abschnitt, ihren eigenen `Address Space`. Für die Firmen hat dies viele Vorteile, beispielsweise müssen weniger Parteien beim Finden des korrekten Abschnitts involviert sein. Aber für die `Zentralrouter` bedeutet es eine immer grössere Datenbank an Zuweisungen. Dieses Problem geht so weit, dass die grossen *Routingtables* inzwischen das physikalische Limit erreichen, was ein einzelner Router verarbeiten kann.

1.3 Zentralisierung

Die Macht in den Händen einiger weniger Kapitalisten und internationaler Unternehmen ist unvorstellbar gross. Einige wenige CEO's, welche nie gewählt, überprüft oder zur Rede gestellt wurden, sind in voller Kontrolle unserer Leben. Egal welcher politischen, wirtschaftlichen oder gesellschaftlichen Ideologie jemand auch folgt, eine solche Abhängigkeit wirft gewisse Fragen und Probleme auf.

Tatsächlich muss man nicht lange suchen, um tatsächliche Gefahren im Zusammenhang mit verschiedenen Technologieunternehmen zu finden. Besonders wenn es um den Datenschutz geht, haben Social-Media Dienste keinen besonders guten Ruf.

Aber neben den ideologischen Fragen und Sicherheitsbedenken gibt es auch noch sehr praktische Probleme in der Art, wie moderne Internet-Dienste implementiert sind.

1.3.1 Datenschutz

Wenn man nicht für etwas zahlt, ist man das Produkt.

Nach dieser Idee ist man für ziemlich viele Firmen ein Produkt. Doch leider muss man realisieren, dass man selbst bei kostenpflichtigen Diensten als Produkt gesehen wird. Denn das Internet hat einen neuen Rohstoff zur Welt gebracht. Wer viele Daten über Menschen besitzt, bekommt binnen kürzester

Zeit Macht.

In ihrer einfachsten Funktion werden Daten für personalisierte Werbung eingesetzt. Damit lassen sich Werbungen zielgerichtet an Konsumenten schicken und der Umsatz, sowohl für Firmen als auch für Anbieter, optimieren.

Werbung ist mächtig und hat einen grossen Einfluss auf den Markt. Aber damit lassen sich lediglich Konsumenten zu Käufen überzeugen oder davon abbringen. Wenn man dies mit dem tatsächlichen Potential in diesen Daten vergleicht, merkt man schnell, wie viel noch möglich ist. Denn die Daten die sich täglich über uns im Internet anhäufen, zeigen mehr als unser Kaufverhalten. Von Echtzeit-Positionsupdates, Anrufe und Suchanfragen bis hin zu privaten Chats und unseren tiefsten Geheimnissen, sind wir meist überraschend unvorsichtig im Umgang mit digitalen Werkzeugen.

Während man davon ausgehen muss, dass Firmen, deren Haupteinnahmequelle Werbungen ist, unsere Daten sammeln und verkaufen, gibt es eine Vielzahl an anderen Firmen, die ebenfalls unsere Daten sammeln, obwohl man von den meisten dieser Firmen noch nie gehört hat. Die Liste der potentiellen Mithörer bei unseren digitalen Unterhaltungen ist nahezu unendlich: Internet-Anbieter, DNS-Dienstleister, CDN-Anbieter, Ad-Insertion-Systeme, Analytics-Tools, Knotenpunkte & Datencenter, Browser, Betriebssysteme, ...

Aus dieser Tatsache heraus lassen sich zwei zentrale Probleme formulieren:

- Selbst für die einfachsten Anfragen im Internet sind wir von einer Vielzahl von Firmen und Systemen abhängig. Dieses Problem wird noch etwas genauer im Abschnitt Abhängigkeit besprochen.
- Wir haben weder ein Verständnis von den involvierten Parteien noch die Bereitschaft, Bequemlichkeit dafür aufzugeben.

1.3.2 Implementierung

Wer sich mit digitaler Infrastruktur auseinander setzt, weiss, dass Webseiten und Apps über *Server* laufen. Diese Server, die eigentlich nichts anderes als optimierte Computer sind, verarbeiten, speichern und liefern die unglaublichen Datenmengen, welche für unser digitales Leben benötigt werden. Mit einem Server, den man selbst mieten oder kaufen kann, steht einem dann nichts im Weg, eine digitale Plattform ähnlich wie *Facebook* oder *Twitter* zu

starten. Doch man realisiert schnell, dass *Facebook* und *Twitter* nicht einfach auf einem *Server* laufen. Denn die benötigten Ressourcen dieser Plattformen übersteigen die Kapazität einzelner *Server* um ein Vielfaches.

Für einen normalen Nutzer mag es so wirken, als ob er *das Facebook* nutzt. In Wahrheit werden seine Anfragen aber von Hunderten oder gar Tausenden von *Servern* verarbeitet. Auch wenn keine offiziellen Daten verfügbar sind, nimmt man an, dass *Facebook* aus über `200'000` *Servern* besteht.

Wenn man sich diese Situation von einer Infrastruktur-Perspektive aus genauer anschaut, sieht man eine interessante Situation vorliegen:

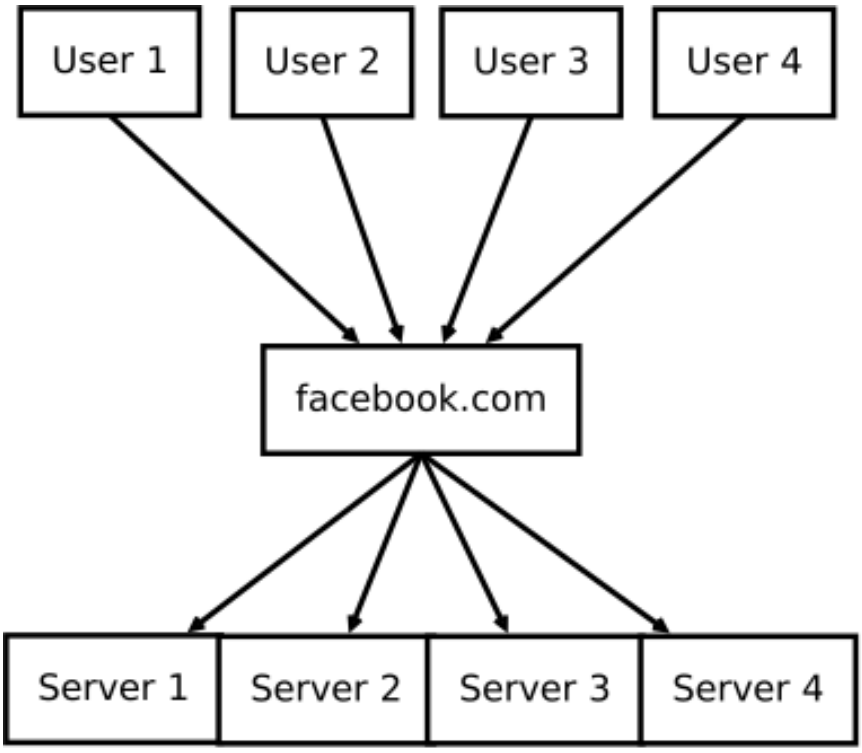

Abbildung 1: Abstrakte Darstellung der Facebook Infrastruktur.

Viele Nutzern greifen also über einen zentralen Knotenpunkt auf viele Server zu. Dieses System, das in bestimmten Situationen schneller ist, sorgt für einen erhöhten Aufwand während der Entwicklung. Anstatt nämlich die

verschiedenen *Server* auf dem gleichen Netzwerk wie die Nutzer zu haben, müssen spezielle Netzwerke und Programme entwickelt und verwaltet werden. Und während die Nutzer weiterhin mit der Illusion *des Facebooks* leben dürfen, haben sie keine Kontrolle über die Sicherheit und Zuverlässigkeit dieser internen Systeme.

Dazu führt dieses System zu einem offensichtlichen Knotenpunkt, durch welchen alle Daten fliessen müssen. Sollten bei diesem zentralen Punkt nun aber Probleme auftreten, lassen sich alle Dienste und Server dahinter nicht mehr erreichen, auch wenn diese eigentlich von den Problemen nicht betroffen wären.

Diese Abstrahierung der internen Infrastruktur führt zur Situation, in der normale Nutzer sich `Facebook` als einzelnen Dienst vorstellen und sich gar nicht bewusst sind, durch wie viele Systeme ihre Daten im Hintergrund fliessen. Denn es garantiert niemand, dass alle Einheiten in einem System wie `Facebook` tatsächlich auch zu `Facebook` gehören, und ob die Daten aller Nutzer tatsächlich bei dieser einen Firma bleiben.

1.3.3 Abhängigkeit

In einem fiktionalen Szenario[1] erklärt *Tom Scott* auf seinem YouTube-Kanal was passieren könnte, wenn eine einzelne Sicherheitsfunktion beim Internetgiganten `Google` fehlschlagen würde. In einem solchen Fall ist es natürlich logisch, dass es zu Problemen bei den verschiedensten `Google`-Diensten kommen würde. Aber schnell realisiert man, auf wie vielen Seiten Nutzer die *Sign-In with Google* Funktion benutzen. Und dann braucht es nur eine böswillige Person um den Administrator-Account anderer Dienste und Seiten zu öffnen, wodurch die Menge an Sicherheitsproblemen exponentiell steigt.

Aber es muss nicht immer etwas schief gehen, um die Probleme zu erkennen. Sei es politische Zensur, *Right to Repair* oder *Net Neutralität*, die grossen Fragen unserer digitalen Zeit sind so relevant wie noch nie.

Während die enorme Abhängigkeit als solche bereits eine Katastrophe am Horizont erkennen lässt, gibt es noch ein konkreteres Problem: Den Nutzern (*den Abhängigen*) ist ihre Abhängigkeit nicht bewusst. Wenn sie sich ihren Alltag ohne `Google` oder `Facebook` vorstellen, denken sich viele nicht viel

[1] Tom Scott: Single Point of Failure `https://youtu.be/y4GB_NDU43Q`, heruntergeladen am 24.05.2020.

darunter. Weniger *lustige Quizfragen* oder Bilder von Haustieren, aber was könnte den schon wirklich Schlimmes passieren?

Während es verständlich ist, dass das Benutzen von `Google` natürlich von `Google` abhängig ist, so versteht kaum jemand, wie viel unserer täglichen Aktivitäten von Diensten und Firmen abhängen, die selbst wieder von `Google` abhängig sind. All dies führt dazu, dass wir auf eine globale Katastrophe zusteuern, die nur darauf wartet, zu passieren.

1.4 Entwicklung

Wer schon mal versucht hat, selbst eine einfache Webseite im Internet zu veröffentlichen, hat schnell gemerkt, wie unglaublich kompliziert der Prozess geworden ist. Natürlich existieren automatisierte Dienste und Anbieter, die den Prozess vereinfachen. Wer aber Wert auf seine Privatsphäre und auf die Verwendung von *open-source* Software legt, muss sich um vieles selbst kümmern. Nicht nur die Auswahl an verschiedenen Programmen kann erschlagend wirken, sondern der Fakt, dass diese untereinander kompatibel sein müssen. Zwar reden wir oft von *einem* Webserver, allerdings sind es tatsächlich viele verschiedene Programme, die alle fehlerfrei miteinander interagieren müssen, um Resultate zu liefern.

Anders als die bereits angesprochenen Probleme, mag dies nicht direkt zu einer Katastrophe und einem Zusammenbruch führen, allerdings existiert die Chance, dass die Komplexität neuer Programme und der Aufwand zur Entwicklung so unglaublich gross werden, dass es sich für einfache Entwickler nicht mehr lohnt, oder das Wissen nicht verfügbar ist, um selbst Dienste im Internet anzubieten.

1.5 Katastrophe

Nachdem nun eine Vielzahl an Problemen angesprochen wurde, muss auch noch die anstehende Katastrophe beschrieben werden. Es gibt verschiedene Gründe, durch welche es zu einer solchen digitalen Katastrophe kommen könnte. Manche sind nur eine Frage der Zeit, andere könnten durch den kleinsten Fehler, sei es technisch oder menschlich, ausgelöst werden.

Es wurden mehrere Engpässe angesprochen. Ob es um Platz im Zentralrouter oder um `IP-Adressen` geht, ist damit zu rechnen, dass es Konflikte, geheime Absprachen und ungerechte Verteilung dieser wichtigen Ressourcen

geben wird. All dies natürlich nur, wenn wir keinen tatsächlichen Krieg starten oder das System vorher zusammenbricht. Und dieser Zusammenbruch muss nicht durch bösswillige Saboteure oder Terroristen kommen. Kleine Tippfehler, Überlastungen oder falsche Abschätzungen können kleinste Fehler auslösen, die sich dann exponentiell über den ganzen Planeten von System zum System verbreiten

Für Viele wird die angesprochene Zentralisierung kaum als Katastrophe wirken, für andere ist die unvorstellbare Macht in den Händen so Weniger bereits Katastrophe an und für sich. Trotzdem ist es wichtig zu realisieren, dass absolute Abhängigkeiten von wenigen, fragilen Systemen selten zu Gutem geführt hat.

1.6 Engine: Orion

Das Internet mit all seinen Facetten und Anwendungen hat viele Probleme. Einige wurden hier bereits angesprochen, andere würden den Rahmen dieser Arbeit sprengen und wieder andere wurden noch gar nicht entdeckt oder als Probleme identifiziert. Aktuell versuchen wir bei jedem neuen Problem eine neue, meist temporäre Lösung zu finden, nur um nichts am System als Ganzem ändern zu müssen.

Langfristig muss aber ein neues System entwickelt werden. Eine grundlegende Neuentwicklung der Art wie wir mit digitalen Systemen interagieren und wie digitale Systeme untereinander agieren. Nur mit einem solchen Wandel können wir die digitale Katastrophe nicht einfach nur verschieben, sondern grundlegend verhindern.

`Engine: Orion` soll eine Rolle in diesem Systemwandel spielen und eine erste Vorlage für ein solches *System der nächsten Generation* bieten. Dabei geht es nicht um ein einzelnes Produkt oder eine Dienstleistung, welche vollständig entwickelt und ohne Mangel ist. Stattdessen sollen verschiedene Komponenten und Systeme entwickelt werden, welche genau gegen die angesprochenen Probleme vorgehen. Mit dem Zusammenschluss dieser Komponenten soll es dann möglich sein, ein neuartiges System zu entwickeln und damit einen Schritt in die Richtung des nötigen Wandels zu gehen.

2 Projekte

Nachdem die zentralen Problemen definiert wurden, ist es an der Zeit, andere Projekte zu analysieren um herauszufinden, ob die Probleme bereits vollständig oder zumindest teilweise gelöst wurden.

Zwar wurden viele Probleme und Gefahren angesprochen, `Engine: Orin` soll sich allerdings nur auf die folgenden beiden Aspekte fokussieren:

- ein Nachrichtensystem zum Senden und Lenken der einzelnen Nachrichten.
- ein System zur Verarbeitung der einzelnen Nachrichten.

Dementsprechend sollen auch andere Projekte, die sich mit diesen beiden Aspekten analysiert werden. Um aber die Wahl der Projekte richtig zu verstehen, muss man die Vision hinter `Engine: Orion` im Blick behalten. Denn es wird schnell klar, dass es für alle beschriebenen Probleme entweder temporäre Lösungen oder einzelne Projekte zur Umgehung der Probleme gibt. Daneben existieren auch grundlegendere Neuentwicklungen bekannter Systeme, welche einzelne Probleme lösen, meist aber andere Ziele haben. Was allerdings kein bekanntes Projekt umzusetzen versucht, ist nicht nur die grundlegende Neuentwicklung zentraler Systeme zur Behebung bekannter Probleme, sondern dazu noch die nahtlose Integration der einzelnen Komponenten für ein durchgehend integriertes, zusammenspielendes System.

Da ein solches Projekt nicht öffentlich existiert, müssen stattdessen die einzelnen Probleme und deren Lösungsversuche angeschaut werden. Dafür wird grundlegend in zwei zentrale Komponenten unterschieden. Die Integration der beiden Teile soll hier nicht besprochen werden, da diese sich hauptsächlich in der Umsetzung zeigt.

2.1 Verteilte Systeme

Viele der Probleme stammen von Designfehlern in unserer globalen Routinginfrastruktur. Mit einer Alternative zu diesem veralteten System würden sich eine Vielzahl von Problemen auf einmal lösen. Der grundlegende Unterschied zwischen *dezentralisierten Systemen* und der aktuellen Umsetzung besteht in der Funktion der Router. Die Problematik der *Zentralrouter* wurde bereits angesprochen. Aber auch kleinere Router führen oft zu Problemen, hauptsächlich durch die absolute Abhängigkeit, die wir zu ihnen entwickelt

haben. Ohne unsere Internetanbieter und die Dienste, die sie uns zur Verfügung stellen, wären wir am Ende, denn nur die Internetanbieter sind in der Lage, ihre Nutzer und Netzwerke mit Routern untereinander zu verbinden.

Auch wenn es viele verschiedene Ideen und Umsetzungen der dezentralisierten Ideen gibt, basieren doch die meisten von ihnen zumindest teilweise auf `Kademlia`, welches also zuerst verstanden werden muss.

2.1.1 Kademlia

Wie geht man also gegen die totale Abhängikeit von Internetanbietern und zentralen Routern vor?
Man kann ja nicht einfach seine eigenen Router aufsetzen und einen alternativen Dienst anbieten. Neben den technischen Schwierigkeiten würde ein solcher Schritt auch überhaupt nichts das eigentliche Problem bekämpfen.

Der Trick, der bei Systemen wie `Kademlia` verwendet wird, ist es, Router vollständig zu eliminieren. Dies ist möglich, indem jedes Mitglied des Netzwerks neben seinen normalen Funktionen gleichzeitig auch noch als Router agiert. Strenggenommen werden in `Kademlia` Router also nicht wirklich eliminiert, lediglich zentrale Router fallen weg.

In einem früheren Abschnitt wurden die Probleme der *Zentralrouter* bereits angesprochen. Wenn jetzt aber jedes Mitglied in einem Netzwerk plötzlich als Router agiert und es keine zentrale Instanz gibt, träfe die Problematik der *Zentralrouter* plötzlich auf alle Server zu. Genau da kommt `Kademlia` ins Spiel. Aber was genau ist `Kademlia` eigentlich?

Laut den Erfindern, *Petar Maymounkov* und *David Mazières*, ist es

> ein Peer-to-peer Nachrichten System basierend auf XOR-Metrik.[2]

Was genau bedeutet das und wie lässt sich eine `XOR`-Metrik für verteilte Datensysteme einsetzen?

Da die einzelnen Server nicht in der Lage sind, Informationen über das komplette Netzwerk zu speichern oder zu verarbeiten, funktionieren `Kademlia`-Systeme grundlegend anders. Anstelle der hierarchischen Anordnung der Router ist jedes Mitglied eines Systems gleichgestellt. dabei kümmert sich

[2]Kademlia: Whitepaper: `https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf`, heruntergeladen am: 30.05.2020.

jedes Mitglied auch nicht um das komplette System, sondern nur um sein direktes Umfeld. Während dies für kleinere Systeme gut funktioniert und vergleichbare Geschwindigkeiten liefert, skaliert es nicht so einfach für grosse Systeme. Genau dafür gibt es die XOR-Metrik.

1. Distanz: Die XOR-Funktion, die in der Informatik an den verschiedensten Orten auftaucht, wird verwendet, um die Distanz zwischen zwei Zahlen zu berechnen. Die Zahlen repräsentieren dabei Mitglieder im Netzwerk und sind je nach Variante im Bereich $0..2^{160}$(*20 Bytes*) oder $0..2^{256}$(*32 Bytes*). Mit einem so grossen Bereich lässt sich auch das Problem der limitierten IP-Adressen lösen. Auch wenn es kein tatsächlich unlimitiertes System ist, so gibt es doch mehr als genug Adressen.

   Wenn mit XOR-Funktionen einfach die Distanz zwischen zwei Zahlen berechnet wird, stellt sich die Frage, wieso nicht einfach die Differenz verwendet wird. Um dies zu beantworten, muss man sich die Eigenschaften der XOR-Funktion etwas genauer anschauen:

   (a) $xor(x, x) = 0$: Das Mitglied mit seiner eigenen Adresse ist zu sich selbst am nächsten. Mitglieder werden hier als Namen für Server in einem Kademlia-System verwendet.

   (b) $xor(x, y) > 0$ wenn $x \neq y$: Die Funktion produziert nie negative Zahlen und nur mit zwei identischen Parametern kann man 0 erhalten.

   (c) $xor(x, y) = xor(y, x)$: Die Reihenfolge der Parameter spielt keine Rolle.

   (d) $xor(x, z) \leq xor(x, y) + xor(y, z)$: Die direkte Distanz zwischen zwei Punkten ist am kürzesten oder gleich kurz wie die Distanz mit einem zusätzlichen Schritt dazwischen, also einem weiteren Sprung im Netzwerk.

   (e) Für ein gegebenes x und eine Distanz l gibt es nur genau ein y für das $xor(x, y) = l$ gilt.

   Aber wieso genau funktioniert dies? Wieso kann man XOR, eine Funktion zur Berechnung der Bit-Unterschiede in zwei Binärzahlen, verwenden, um die Distanz zwischen zwei Punkten in einem verteilten Datensystem zu berechnen?

Um dies zu verstehen, hilft es, sich das System als umgekehrten Baum vorzustellen. Untergeordnet zum zentralen Punkt zu oberst stehen alle Mitglieder im System. Mit jeder weiteren Abzweigung zweier Teilbäume halbiert sich die Anzahl. Man wählt am einfachsten zwei Abzweigungen pro Knoten, da sich damit die Werte direkt als Binärzahlen darstellen lassen, wobei jeder Knotenpunkt einfach eine Stelle in der langen Kette aus 0 oder 1 darstellt. Der ganze Baum sieht dann wie folgt aus[3]:

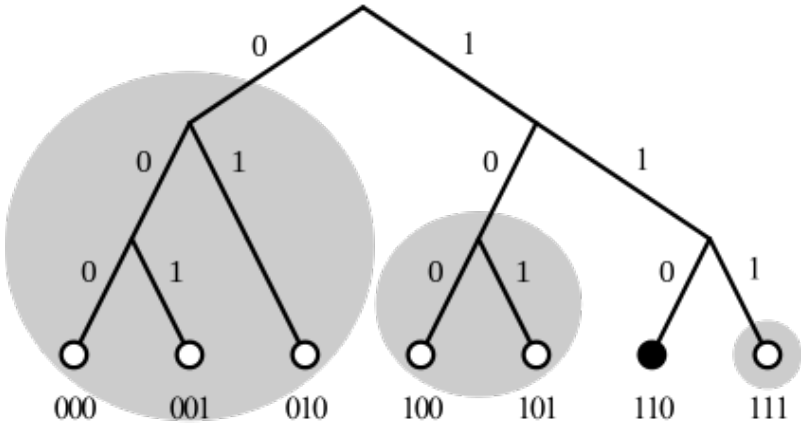

Abbildung 2: Beispielhafte Darstellung eines einfachen Kademlia-Systems

Mit dieser Sicht auf das System beschreibt die `XOR`-Funktion die Anzahl der unterschiedlichen Abbiegungen im Baum und somit die Distanz. Zwar mag es auf den ersten Blick nicht intuitiv wirken, wieso man `XOR` anstatt einfach der Differenz verwendet, allerdings funktioniert die Funktion mit Binärzahlen in einem solchen Baum einiges besser.

2. Routing-Table: In einem `Kademlia`-System hat jedes Mitglied eine gewisse Anzahl anderer Mitglieder, mit denen es sich verbunden hat. Da `Kademlia` ein sehr umfangreiches, kompliziertes Protokoll und System beschreibt, sollen hier nur einige zentrale Funktionen besprochen werden, die für diesen ersten Prototypen von `Engine: Orion` relevant sind. Besonders beim `Routing Table` lassen sich einige Abschnitte

[3]Wikipedia: Kademlia `https://en.wikipedia.org/wiki/Kademlia`, heruntergeladen am: 30.05.2020.

weglassen, welche zwar für die Optimierung und Skalierung eines Systems wichtig, allerdings für das Analysieren eines einfachen, kleinen Systems wie `Engine: Orion` irrelevant sind.

Einfach formuliert speichert der `Routing Table` die verbundenen Mitglieder. Eine eingehende Nachricht wird dann mithilfe dieser Liste, sowie der `XOR`-Metrik ans richtige Ziel geschickt. Um das System zu optimieren und die Anzahl der benötigten Sprünge klein zu halten, wird ein spezielles System verwendet, um zu entscheiden, welche Mitglieder im `Routing-Table` gespeichert werden sollen:

(a) Sehr nahe an sich selbst (in der obigen Darstellung also: wenige Sprünge im Baum) werden alle Mitglieder gespeichert.

(b) Je weiter weg sich die Mitglieder befinden, desto weniger werden gespeichert.

Die optimale Anzahl der gespeicherten Mitglieder hängt von den Zielen und Ansprüchen an das System ab. Grundlegend muss man die Frage beantworten, mit wie vielen Unterbäumen Verbindungen gehalten werden sollen. Zwar mag dies etwas abstrakt wirken, es lässt sich aber mit dem eben eingeführten Modell eines umgekehrten Baumes gut erklären:

- In der obersten Ebene trennt sich der Baum in zwei vollständig getrennte Teile, was sich mit jeder weiteren Ebene wiederholt.
- Die einzige Möglichkeit vom einen *Ende* des Baums zum anderen zu kommen, ist über den obersten Knoten. Um also in die andere Hälfte zu kommen, braucht man mindestens eine Verbindungsstelle in der anderen Hälfte.
- Deshalb braucht ein `Routing-Table` nicht nur kurze Verbindungen zu nahen Punkten, sondern auch einige wenige Verbindungen zu Mitgliedern in der anderen Hälfte.
- Mit nur einer weit entfernten Adresse hat man eine Verbindung in *eigene* und die *andere* Hälfte. Hat man zwei solche Verbindungen auf die andere Seite hat man schon Verbindungen in jeden Viertel des Baumes.
- Man muss also entscheiden, wie fein man die andere Hälfte aufteilen will. (Eine genaue Unterteilung bedeutet wenige Sprünge

aber grosse Routing-Tables, eine grobe Unterteilung genau das Umgekehrte).

Zwar hat ein vollständiges Kademlia-System noch komplexere Elemente, wie k-Buckets, welche den Routing-Table optimieren, allerdings sind diese für die grundlegende Funktionsweise des Systems irrelevant.

Die dynamische Regulation des Routing-Tables muss allerdings noch erwähnt werden:

- Sobald die definierte Maximalgrösse erreicht ist, werden keine neuen Verbindungen akzeptiert.
- Zwar können existierende Einträge durch Neue ersetzt werden, allerdings werden dabei nicht alte, sondern inaktive Einträge entfernt. Für ein Kademlia-System werden also auch Mechanismen benötigt, um die Zustände aller Verbindungen periodisch zu überprüfen.

2.1.2 BitTorrent

Dezentralisierung hat viele Vorteile und muss langfristig flächendeckend eingesetzt werden. Aktuell sind die meisten Industrien und Produkte noch nicht so weit. Trotzdem gibt es einige Anwendungen und Gruppen bei denen solche Systeme bereits seit Jahren Verwendung finden.

Beispielsweise im Zusammenhang mit *(mehr oder weniger legalen)* Verbreiten von Materialien wie Filmen oder Musik wird eines der grössten global verteilten Systeme eingesetzt. Natürlich gibt es hunderte von verschiedenen Programmen, Ideen und Umsetzungen, wobei die meisten Nachfolger von Napster sind.

Im preisgekrönten Film *The social network* erhält man Einblick in den Lebensstil von Sean Parker, einem der Gründer von Napster. Es mag überraschen, wie jemand wie Parker, der nur wenige Jahre zuvor mit Napster die komplette Musikindustrie in Unruhe gebracht hatte, eine so zentrale Rolle in Facebook, einem der zentralisiertesten Megaunternehmen der Welt, einnehmen konnte.

Auch wenn es noch nicht *vollständig* dezentralisiert ist, erlaubte es Napster Nutzern, Musik über ein automatisiertes System mit anderen Nutzern zu teilen und neue Titel direkt von den Geräten anderer Nutzer herunterzuladen. Dabei gab es allerdings immer noch einen zentralen Server, der die Titel sortierte und indizierte.
Napster musste am Ende abgeschaltet werden, nachdem die Klagen der Musikindustrie zu belastend wurden. Auch wenn das Produkt abgeschaltet wurde, liess sich nichts mehr gegen die Idee unternehmen.

Über viele Iterationen und Generationen hinweg wurden die verteilten Systeme immer weiter verbessert, jegliche zentrale Server entfernt und in die Hände immer mehr Nutzer gebracht. Heute läuft ein Grossteil des Austauschs über BitTorrent.
BitTorrent baut auf der gleichen grundlegenden Idee wie Napster auf: Nutzer stellen ihren eigenen Katalog an Medien zur Verfügung und können Inhalte von allen anderen Mitgliedern im System herunterladen. Anders als Napster gibt es bei BitTorrent keine zentrale Komponente, stattdessen findet selbst das Indizieren und Finden von Inhalten dezentralisiert statt[4]. Dafür wird über das Kademlia-System aktiv bekannt gegeben, wer welche Inhalte zur Verfügung stellt, wobei einzelne Mitglieder speichern können, wer die gleichen Inhalte anbietet. Neben Dezentralisierung und Sicherheit lassen sich über BitTorrent tatsächlich gute Geschwindigkeiten erreichen, da sich Inhalte von mehreren Anbietern gleichzeitig herunterladen lassen. Da es sich bei BitTorrent eigentlich um ein grosses Dateisystem handelt, lassen sich direkt die SHA1-Hashwerte der Inhalte als Kademlia-Adressen verwenden.

2.1.3 CJDNS

Im CJDNS-Whitepaper[5] werden viele der gleichen Probleme erwähnt, wie auch für Engine: Orion angesprochen wurden. Die grundlegenden Ideen und Lösungsansätze, die besprochen werden, ähneln in vielerlei Hinsicht den Ideen und Prinzipien hinter Engine: Orion. CJDNS, das sich selbst als

> an encrypted IPv6 network using public-key cryptography for address allocation and a distributed hash table for routing

beschreibt, ist ein beliebtes *open-source* Projekt, mit mehr als 180 Mitentwicklern und Tausenden von Nutzern. Es basiert ebenfalls auf Kademlia

[4] BitTorrent: Mainline DHT: https://www.cs.helsinki.fi/u/lxwang/publications/P2P2013_13.pdf, heruntergeladen am: 4.06.2020.

[5] CJDNS - Whitepaper: https://github.com/cjdelisle/cjdns/blob/master/doc/Whitepaper.md, heruntergeladen am: 3.06.2020.

und ähnlich wie bei `BitTorrent` wird grosser Wert auf Sicherheit und Verschlüsselung gelegt.

Wer den Code von `CJDNS` etwas genauer anschaut realisiert schnell, welche grundlegenderen Ziele `CJDNS` verfolgt. Tatsächlich soll mit `CJDNS` langfristig ein physikalisch unabhängiges Netzwerk entstehen. Das Whitepaper redet von der Freiheit der Nutzer, eigene Kabel und eigene Infrastruktur zu verlegen.

Es mag auf manche nach digitaler Isolation und Abschottung klingen, aber wer tatsächlich konsequent alle Probleme von Grund auf angehen will, muss sich auch Gedanken über die darunterliegende physikalische Infrastruktur machen. Dies ist allerdings ein Schritt der mit `Engine: Orion` (*noch*) gemacht werden soll.

2.2 Nachrichtenverarbeitung

In einem nächsten Schritt muss man sich fragen, wie die Nachrichten tatsächlich verarbeitet werden sollen. Welche Programme, Programmiersprachen und Recheneinheiten sollen Zugang haben, und wie sollen Entwickler tatsächlich mit dem System interagieren?

Im Laufe der Zeit gab es verschiedene Trends, wobei die meisten auf immer kleinere Einheiten der Berechnung zusteuerten. Als neuartiges System soll `Engine: Orion` aber nicht einfach dem aktuellen Trend folgen. Es sollen stattdessen die verschiedenen Schritte und Optionen sollen analysiert werden, sodass daraus dann ein zukunftssicheres System entstehen kann.

2.2.1 Betriebssysteme

Computer werden immer leistungsfähiger und sind immer mehr verbunden. Mit der neu gewonnen Leistungsfähigkeit entstanden immer komplexere Programme und somit auch immer mehr Funktionen und Fähigkeiten, die dem Computer übergeben werden können. Die meisten Personen vertrauen ihren Computern hoch sensitive Informationen an, die keine unbeteiligte dritte Person erhalten sollte. Diese Informationen sind aber sowohl für Kriminelle und als auch für viele Werbebetreiber und moderne Internetfirmen von grossem Interesse und oftmals ist es äusserst einfach, an die Daten zu kommen. Um dagegen vorzugehen wurden in den letzten Jahrzehnten verschiedene Programme entwickelt, die sich darauf spezialisiert haben, verschiedene Da-

ten und Programme zu isolieren, um die Sicherheit und Integrität der Daten zu sichern.

Angefangen hat diese Technologie mit der Entwicklung des Betriebssystems. Microsoft wollte Ende der 80er Jahre sicher gehen, dass das Betriebssystem mehr Rechte hat als die Programme, die auf dem Computer laufen. Damit diese Zuweisung von Rechten funktioniert, braucht das Betriebssystem auch Hardware, die dies unterstützt. Die zwei wichtigsten Komponenten ist die *Memory Management Unit* (`MMU`) und die Fähigkeit den Modus der CPU zu wechseln. Die `MMU` wurde von Intel schon seit längerer Zeit in ihren Chips eingebaut und regelt die Konvertierung von Adressen, die das Programm vorgibt, zu tatsächlich existierenden Adressen. Diese Konvertierung muss passieren, da jedes Programm davon ausgeht, dass es bei der Adresse 0 startet. Die zweite wichtige Aufgabe ist die Validierung der Adressen. Wenn ein Programm auf eine Adresse zugreift, die nicht dem Programm gehört, muss das Betriebssystem aufgerufen werden, damit sichergestellt werden kann, dass das Programm niemals Daten von anderen Programmen bekommen kann.

Die andere, bereits angesprochene wichtige Funktion ist das Wechseln des Modus. Wenn ein Betriebssystem geladen ist, das es erlaubt mehrere Programme laufen zu lassen, muss die CPU mehrere Programme kurz nacheinander abarbeiten. Da die Programme aber meistens verschiedene Rechte haben, muss die CPU die Rechte immer wieder abändern. Diese Funktion wollten die Hardware-Hersteller am Anfang nicht unterstützen und deshalb wurden die Ingenieure bei Microsoft kreativ. Da sie die Wichtigkeit der Abkapselung erkannten, haben sie keine Mühe gescheut und ein höchst Ausgeklügeltes System entwickelt. Wenn ein anderes Programm ausgeführt werden soll, welches mit anderen Rechten läuft, übernahm das Betriebssystem die Kontrolle und führte ein Befehl aus, der kontrolliert den Prozessor zum Neustart gezwungen hat. Mit diesem unkonventionellen Trick wurde die Abkapselung in der Informatik salonfähig gemacht.

2.2.2 Cloud-Functions

Im Laufe der Zeit waren vollständige Betriebssysteme nicht mehr genug. Man wollte kleinere Einheiten, die sich einfacher verwalten liessen. Aus diesem Verlangen heraus entstanden `Docker` und ähnliche Container-Systeme, die auch heute noch überall zu finden sind. In der Fortführung des dauerhaften Trends der Verkapselung und Isolierung ging man aber noch einen Schritt weiter:

Anstelle der isolierten Betriebssysteme oder Container ist es auch möglich, einzelne Funktionen anzubieten. Natürlich haben diese nicht den Funktionsumfang und die Möglichkeiten eines vollständigen Webservers oder Containers, aber Google und andere Cloud-Anbieter erkannten, dass ein Grossteil der einfachen Webserver und API's ähnliche Funktionen übernahm. Im Falle von Google wurde daraufhin ein neues System entwickelt, welches die eigene Programmiersprache Golang mit den eigenen Datenbanksystemen wie Firestore verband und es Nutzern erlaubten, einzelne Funktionen anzubieten. Ein System welches unter dem Namen Google Cloud Functions[6] bekannt ist.

Diese kleinstmöglichen Einheiten haben verschiedene Vorteile:

- Nutzer können für jeden Aufruf einzeln bezahlen, müssen also nicht in kompletten Servern über ihre Kosten nachdenken.
- Die Verwaltung von Docker-Images und virtuellen Maschinen fällt komplett weg.

Wer spezielle Ansprüche oder komplexe Systeme will, wird mit Cloud Functions nicht gut versorgt sein, aber sie ermöglichen es, kleine Funktionen mit dem geringstmöglichen Aufwand umzusetzen.

3 Konzept

Nachdem nun die zentralen Probleme festgehalten und einige der aktuellen Bemühungen genauer angeschaut und verglichen wurden, müssen die Ziele und Prinzipien für Engine: Orion in ein nutzbares System umgesetzt werden.

Zuerst ist es wichtig, sich nochmals an die grundlegende Idee hinter Engine: Orion zu erinnern:

- Eine grundlegende Änderung in der Art, wie wir mit anderen Personen und digitalen Systemen interagieren, ist von Nöten.
- Engine: Orion soll weder alle bisherigen Systeme ersetzen, noch dem Status Quo den Kampf ansagen. Stattdessen soll Engine: Orion vorzeigen, was möglich ist und anhand einer beispielhaften Umsetzung einsetzbare Produkte präsentieren.

[6] Google: Cloud Functions: https://cloud.google.com/functions, heruntergeladen am: 6.06.2020.

Mit diesen grundlegenden Zielen müssen die einzelnen Komponenten des Systems definiert und umgesetzt werden. Dafür soll sowohl zur Simplifizierung der Entwicklung, als auch zur späteren Erweiterbarkeit ein modulares Komponentensystem verwendet werden. Alle Teile des Systems sollen über ein einheitliches Protokoll miteinander verbunden sein.

Neben den Prinzipien des Projektes, gibt es zusätzliche Prinzipien der Entwicklung, an die sich alle Teile des Systems halten sollten:

- Der gesamte Code soll jederzeit öffentlich verfügbar und gut dokumentiert sein.
- Die einzelnen Komponenten sollen nur von *open-source* Programmen und Projekten abhängig sein.

4 Prototyp

Im nächsten Kapitel sollen die einzelnen Komponenten und Programme dank welchen `Engine: Orion` funktioniert, genau erklärt werden. Da das ganze System aber ein komplexes Zusammenspiel verschiedener Programme, Programmiersprachen und Geräten benötigt, muss als erstes ein grobes Verständnis über das System als Ganzes Vorhanden sein.

Mit den nächsten Abschnitten soll der Prototyp als ganzes System erklärt werden. Dafür soll eine `Chat-Demo` verwendet werden, da diese neben einem praktischen Einsatzgebiet keine zusätzliche Komplexität ins System einführt. Aller Code für diese Demo ist auf `GitHub`[7] zu finden. Es ist also möglich, selbst diese Demo auszuprobieren, da alle Programmiersprachen und Abhängigkeiten *open-source* und frei sind (Das System wurde nur auf Linux- / Unix-Systemen getestet und hat möglicherweise Probleme auf anderen Plattformen). Zwar ist die Demo an sich nichts besonders, sie zeigt allerdings perfekt die Stärken und Einsatzmöglichkeiten für `Engine: Orion` und ähnliche Systeme. Dabei sollen die drei wichtigen Perspektiven einzeln erklärt werden:

- Als erstes soll die Sicht eines Nutzers, unabhängig von dessen Wissensstand, gezeigt werden.

[7] Engine: Orion - GitHub: `https://github.com/EngineOrion`, heruntergeladen am: 9.06.2020.

- In einem nächsten Schritt soll die Sicht eines potentiellen Entwicklers[8] für das System dargelegt werden.
- Da manche Komponenten so weit abstrahiert sind, dass sie sowohl für Nutzer, als auch für Entwickler unsichtbar sind, muss die *Sicht* der Daten und ihr Weg durch das System festgehalten werden.

4.1 Nutzer

In jedem modernen Informationssystem geht es um Nutzer und um ihre Aktivitäten, daher ist es nur logisch, `Engine: Orion` zuerst aus ihrer Perspektive zu betrachten. In dieser beispielhaften Erklärung geht es um einen online Chatraum, in welchem sich verschiedene Nutzer miteinander über Nachrichten austauschen können. Als erstes gibt es zwei sehr wichtige Feststellungen über die Demo, sowie über das System als Ganzes:

- In seiner aktuellen Fassung läuft das verteilte Datensystem nur auf *Servern*, nicht bei jedem einzelnen Nutzer. Dadurch fällt beinahe jegliche Komplexität bei den Nutzern weg ab und liegt stattdessen bei den *Server*-Betreibern.
- `Engine: Orion` schreibt nicht vor, wie die *last-mile-delivery*, also die letzte Strecke der Daten von einem Server zu den Nutzern, passieren soll. Daher ist es möglich, Webseiten, `HTTP-API`'s oder ähnliches in das System einzubinden, ohne den Datenfluss des verteilten Datensystems oder anderer Komponenten zu beeinflussen. Im Falle des Chats bedeutet das, dass Nutzer den Dienst über eine einfache Webseite verwenden können, welche im Hintergrund Websockets verwendet.

Wie verhält sich eine Seite basierend auf `Engine: Orion` für einen normalen Nutzer?
Genau gleich wie jede andere Seite auch.
Natürlich hat dieses System auch einige Nachteile:

- Nutzer müssen sich im aktuellen System gezielt mit einzelnen Servern verbinden, von ihrer Perspektive aus ist `Engine: Orion` also nicht richtig dezentralisiert. Dazu können Nutzer nicht einfach einen beliebigen Server wählen, denn sie brauchen einen, der das spezielle Interface, in diesem Falle Websockets, unterstützt. (Zwar wäre es möglich, das System umzustrukturieren, sodass einzelne Nutzer sich direkt verbinden

[8] Die Verwendung der männlichen Form soll im gesamten Dokument für den generischen Maskulin stehen.

können, allerdings würde dies eine Vielzahl neuer Probleme einführen, beispielsweise eine viel höhere `Churn-Rate`[9].)

- Nichts hält einen potentiellen Anbieter einer Seite davon ab, `Cookies`, Tracking oder ähnliche Probleme des *alten Nets* einzubauen.

Natürlich ist dieses System bei weitem nicht perfekt und in einer späteren Iteration von `Engine: Orion` wäre es denkbar, direkte Zugriffe von Servern und von einzelnen Nutzern zu erlauben. Trotzdem ist die aktuelle Umsetzung in Sachen Bedienungsfreundlichkeit für normale Nutzer kaum zu schlagen.

4.2 Entwickler

In diesem nächsten Abschnitt soll die Perspektive der Entwickler angeschaut werden. Mit Entwicklern sind dabei die gemein, die eigene *Server* für `Engine: Orion` betreiben und Dienste basierend auf der Infrastruktur anbieten. Eben wurde bereits die Modularität und Vielseitigkeit des Prototypen angesprochen. Diese hat natürlich hauptsächlich Vorteile für die Entwickler und erlaubt es ihnen, bereits existierende Systeme sowie neue Komponenten, einfach zu integrieren. Im Falle dieses Beispiels muss ein Entwickler lediglich die tatsächlichen Chat-Funktionen programmieren und richtig integrieren. Zum Glück macht dies der Prototyp sehr einfach, auch wenn man sich an neue Technologien und Werkzeuge gewöhnen muss.

Die Verarbeitung der Nachrichten, also die eigentliche Chat-Funktion, findet auf einem *Server* im Netzwerke statt. Dabei spielt es keine Rolle, ob Nutzer direkt mit diesem Mitglied (Mitglied ist der `Engine: Orion` Begriff für einen *Server* oder ein *Mitglied* eines Netzwerks) interagieren, denn das Netzwerk kümmert sich von selbst um die Zustellung der Nachrichten an die verantwortliche Stelle. Ein Entwickler muss also auf seinem eigenen oder einem beliebiges Mitglieds seinen Code, mit einem Programm namens *Container* registrieren. Dafür wird ein gleichnamiges Programm[10] benötigt. Dieses kümmert sich um das tatsächliche Zustellen der Nachrichten und bietet zusätzlich verschiedene Schnittstellen für eigene Programme. Diese Programme werden in der speziell dafür entwickelten Sprache `NET-Script` geschrieben,

[9] Churn in Kademlia Systemen: `https://people.inf.ethz.ch/troscoe/pubs/usenix-cr.pdf`, heruntergeladen am: 30.05.2020.

[10] Engine: Orion - Container: `https://github.com/EngineOrion/container` heruntergeladen am: 9.06.2020.

welche neben einfachen Schnittstellen mit dem restlichen System auch verschiedene Funktionen und Hilfestellungen zum Verarbeiten von Nachrichten anbietet.
Der Code eines Entwicklers wird also mit dem *Container*-Programm verarbeitet und ausgeführt. Um welche Komponenten muss sich ein Entwickler sonst noch kümmern?

Es wurde bereits ein Programm zur Verwaltung des Netzwerks angesprochen. Mit diesem sollte ein Entwickler allerdings nie direkt interagieren. Stattdessen steht dazwischen noch ein lokaler *Router* namens `hunter`[11], welcher Nachrichten den passenden Programmen und Schnittstellen zuweist.

4.3 Daten

Da bereits an verschiedenen Stellen Komponenten im System angesprochen wurden, die weder mit Nutzern, noch mit Entwicklern direkt interagieren sollen, muss zum Schluss noch das ganze System aus der Perspektive der Daten, also aus der Perspektive einer einzelnen Chat-Nachricht angeschaut werden. Dabei soll der gesamte Weg einer Nachricht zwischen zwei Nutzern betrachtet, und alle beteiligten Systeme grob erklärt werden.

Der Weg einer Chat-Nachricht beginnt am Anfang, dort wo ein Nutzer die Nachricht eintippt und danach *Eingabe* drückt. Wie landet diese Nachricht nun bei einem oder mehreren Nutzern?

- Zuerst muss sie an einen Server im *System* übertragen werden. Im Falle dieses Beispiels geschieht dies über Websockets, allerdings schreibt `Engine: Orion` nicht vor, über welches Protokoll oder Medium Nutzer kommunizieren. Durch die hohe Modularität und Isolation der einzelnen Komponenten ist es genauso möglich, ein lokales Programm, zum Beispiel ein Videospiel oder Daten-Austausch-Programm, zu integrieren. Solange das Protokoll richtig implementiert ist, und ein Programm mit den nötigen Komponenten interagieren kann, ist es in der Lage, an `Engine: Orion` angeschlossen zu werden.
  In diesem Falle wird die Nachricht über eine Komponente namens `websocket-or`[12] entgegen genommen. Dieser ist auch gleichzeitig da-

[11] Engine: Orion - Hunter: `https://github.com/EngineOrion/hunter`, heruntergeladen am: 9.06.2020.
[12] Engine: Orion - Websocket-OR: `https://github.com/EngineOrion/websocket-or`, heruntergeladen am: 9.06.2020.

für verantwortlich, die HTML-Seite zu liefern, auch wenn diese Kombination nicht zwingend ist.

- Wie jeder Bestandteil muss `websocket-or` ein Kommunikationsprotokoll über `Unix-Domain-Sockets` unterstützen. Dabei wird die Nachricht, sowie einige Metadaten in JSON-Format, über einen Domain-Socket (oder auch `IPC-Socket` genannt), an das nächste Element weitergegeben. Zwar wäre es möglich, die Daten direkt an das verteilte Datensystem zu geben, aber es wird noch ein Zwischenschritt eingeführt. Dieser ermöglichst es später, zusätzliche Programme an das System anzuschliessen und sorgt dafür, dass die passenden Daten bei den richtigen Applikationen landen. Über `hunter`, den lokalen Router eines jeden `Orion`-Servers, wird die Chat-Nachricht, die lokal nicht weiterverarbeitet werden kann, an das nächste Glied in der Kette gegeben, ebenfalls über einen `IPC-Socket`.

- Da im lokalen Server niemand die Nachricht weiterverarbeiten konnte, wird sie an das verteilte Datensystem übergeben, um fähiges Mitglied zu finden, das in der Lage ist, die Nachricht zu verarbeiten. `Shadow`[13], das verteilte Datensystem für `Engine: Orion`, überträgt dann die Nachricht an das passende Mitglied aufgrund der Metadaten in der Nachricht.
  Auf jedem *Server* wird wieder `hunter` verwendet, um zu überprüfen, ob das Ziel der Nachricht, also der eigentliche *Chat-Server* (der Begriff *Chat-Server* beschreibt dabei keinen physikalischen Server, sondern ein spezielles Element für `Engine: Orion`: Ein *Container*), lokal registriert ist.

- Sobald das richtige Mitglied gefunden wurde, wird wieder ein `IPC-Socket` verwendet, um die Daten zu übertragen. Auch hier dient `hunter` wieder als Überträger und Router. In diesem Fall wird die Nachricht an ein weiteres Programm übertragen, welches sich um die tatsächliche Verarbeitung kümmert. Mit einem passenden Programm speziell für den Chat wird die Nachricht an alle verbundenen Nutzer weitergeleitet. Im Container wird eine dafür entwickelte Programmiersprache namens `NET-Script` verwendet, welche einfache Integration mit dem übrigen Komponenten bietet.

[13] Engine: Orion - Shadow: `https://github.com/EngineOrion/shadow`, heruntergeladen am: 9.06.2020.

- Nachrichten, die von den Containern wieder bei `hunter` landen, werden aufgrund ihrer Meta-Daten an die passenden Komponenten oder wieder über das verteilte Datensystem verbreitet, wobei der ganze Ablauf rückwärts abläuft.

5 Komponenten

In diesem Abschnitt sollen die Fähigkeiten und Leistung jedes einzelnen Programms in `Engine: Orion` behandelt werden. Dabei geht es aber weniger um ihr Zusammenspiel und ihre Rolle im grossen Ganzen. Stattdessen soll der Aufbau und die Funktionalität der Komponenten als unabhängige Programme besprochen werden.

Es kann hilfreich sein, die obige Anschauung des Systems aus der Perspektive einer Chat-Nachricht zu verwenden. Die Abschnitte hier sind allerdings nicht in dieser Reihenfolge, da die besprochene Chat-Applikation lediglich eine potentielle Anwendung darstellt und `Engine: Orion` mit Modularität und Erweiterbarkeit als Priorität entwickelt wurden. In den folgenden Abschnitten soll es aber um Code und Mechanismen der Programme gehen. Zwar wird alles möglichst von Grund auf erklärt, allerdings wird gewisse technische Vorkenntnisse als Grundlage angenommen.

5.1 Hunter

Herzstück einer jeden `Engine: Orion` Instanz ist sich `hunter`. Jede andere Komponente ist eigentlich optional. Man könnte das System ohne Datensystem, ohne Verarbeitung und ohne Inputs einsetzen. Aber ohne `hunter` geht es nie. `Hunter` ist auch die erste Applikation, die gestartet werden muss und es kümmert sich um die Verbindung aller anderen Teile. Da `hunter` in jeder Nachricht für jeden Komponenten involviert ist, müssen zwei zentrale Bedingungen erfüllt werden:

- Geschwindigkeit: `hunter` muss schnell sein. Sowohl für die Übertragung von Nachrichten über Sockets, als auch für das Lesen der verschiedenen Konfigurationen über ein `CLI` (*Command Line Interface*) muss die Geschwindigkeit bedacht werden. Eine Verzögerung in `hunter` könnte sich durch das ganze System hinweg vervielfachen und selbst kleinste Abweichungen könnten grosse Konsequenzen für das System als Ganzes haben.

- Zuverlässigkeit: Ähnlich wie die Geschwindigkeit könnten Fehler, Abstürze oder Unsicherheiten in `hunter` grosse Probleme für das ganze System bedeuten.

Mit diesen beiden zentralen Zielen musste `hunter` auf zuverlässigen, schnellen und sicheren Technologien aufbauen. Da `hunter` zusätzlich als installiertes Programm verwendet werden soll, war `Rust` die beste Option. Durch das integrierte *Ownership* und *Borrowing* System in `Rust` kann `hunter` zuversichtlich als sicheres System in die restlichen Komponenten integriert werden.

`Hunters` Funktionen lassen sich in zwei Bereiche aufteilen, wobei die Unterscheidung durch Argumente beim Start des Programms gemacht wird. Entwickler müssen also `hunter` nur einmal installieren und starten, die interne Interaktionen finden dann voll automatisch statt.

5.1.1 CLI

Um die Integration mit anderen Komponenten möglichst einfach zu machen, werden verschiedene Funktionen über ein *Command Line Interface* angeboten. Beispielsweise der Test, ob ein Eintrag lokal registriert ist geschieht durch das Auslesen der lokalen Konfiguration, welche dann mit den gegebenen Parametern abgeglichen wird. Diese Art der Interaktion erlaubt es auch, weitere Programme in `hunter` und `Engine: Orion` zu integrieren, ohne für diese die besonderen `Socket`-Protokolle zu implementieren. Aktuell sind die Funktionen der automatischen Verwaltung noch stark limitiert, diese sollte aber zu einem späteren Zeitpunkt mit der Zuweisungslogik von `shadow` verbunden werden.

5.1.2 Sockets

Für das Senden und Empfangen der tatsächlichen Nachrichten im System eignet sich die `CLI` nicht, da dafür aktive Prozesse benötigt werden. Zwar würde sich das theoretisch mit `daemons` oder ähnlichen Prozessen lösen lassen, allerdings würde diese Option unnötige Komplexität ins System bringen. Stattdessen lässt sich `hunter` mit dem Argument

```
hunter start
```

starten, womit dann alle `IPC`-Sockets gestartet werden. Sobald eine Nachricht eingeht, werden die gleichen Funktionen wie für das `CLI` verwendet, um das Ziel der Nachricht zu finden. Danach wird die Nachricht an den passenden `Socket` gesendet. Da jeder Socket einen eigenen Thread übernimmt,

müssen die Nachrichten zwischen den verschiedenen Threads gesendet werden. Dafür werden `Rust-MPSC-Channels` verwendet. Diese erlauben die schnelle Übertragung der Nachrichten an den zuständigen Komponenten.

5.2 Shadow

Auf den ersten Blick mag der Name `shadow` für ein verteiltes Datensystem etwas seltsam wirken. Aber wenn man sich die bisherigen Erwähnungen von `shadow` nochmals anschaut, merkt man schnell, dass es weder in der Nutzer-Perspektive, noch in der Entwickler-Perspektive eine Rolle spielte. Erst während der dritten Analyse, also aus der Sicht der Daten spielte `shadow` tatsächlich eine Rolle. Für die Nutzer und Entwickler, die mit `Engine: Orion` interagieren, ist `shadow` im Schatten verborgen.
Für viele der Ziele und Prinzipien wäre `shadow` als komplett eigene Lösung nicht nötig gewesen. Über das *normale* Internet, oder über eine bereits existierende Lösung, hätte sich eine ähnliche Funktionalität erreichen lassen. Allerdings erlaubt `shadow` die nahtlose Integration und Verbindung verschiedener Server und geht gleichzeitig das Problem der Zentralisierung an.

Während der Entwicklung dieses verteilten Datensystems mussten zwei zentrale Kriterien erfüllt werden:

- einfache Interaktion mit Netzwerken über `TCP` oder `UDP`
- schnelle Entwicklung zum Testen neuer Funktionen und Entwickeln eines Prototypen.

Mit `Elixir` existiert eine Sprache, die genau dies ermöglicht. Basierend auf der `Erlang-Virtual-Machine`, welche seit Jahrzehnten globale Telekommunikation ermöglicht, verbindet `Elixir` einen `Ruby` und `Python` ähnlichen Syntax mit dem bekannten Nebenläufigkeits-Modell von `Erlang`, welches in den letzten Jahren die Entwicklung der Nebenläufigkeitsmodelle in Sprachen wie `Go` (Golang) oder `Rust` inspiriert hat. Mit einer aktiven Gemeinschaft, welche sowohl aus Neuzugängern wie auch aus Veteranen der Telekommunikationsindustrie und Netzwerk-Entwicklung besteht, ist `Elixir` eine Sprache die viel Potential hat und sich aktuell in den GitHub-Umfragen unter den *meist geliebten* Programmiersprachen überhaupt befindet.

Neben den oben genannten Ansprüchen an die Entwicklung sollten alle `Engine: Orion` Komponenten mit möglichst wenig externen Abhängigkeiten auskommen. Durch eine grosse Standardbibliothek und Zugang zu allen Funktionen

und Hilfsprogrammen für das Erlang-Ökosystem kommt shadow für alle Netzwerklogik komplett ohne externe Programme aus. Für die tatsächlichen Interaktionen mit anderen Servern über TCP werden Sockets verwendet, welche vom Betriebssystem, beziehungsweise vom Linux-Kernel, zur Verfügung gestellt werden. Als einzige tatsächliche Abhängigkeit für den Code wird ein JSON-Parser importiert, welcher für Nachrichten und Konfigurationsdaten verwendet wird.

In einem ersten Schritt muss die allgemeine technische Funktionsweise von shadow angeschaut werden.Danach soll der innere Aufbau noch etwas genauer erklärt werden. Für diesen Schritt sind gewisse Grundkenntnisse über Elixir hilfreich. Anleitungen sind aber nicht schwer zu finden[14]. Grundsätzlich kümmert sich shadow um das Verwalten von Verbindungen zwischen verschiedenen Servern und das Senden von Nachrichten zwischen ihnen. Solche Verbindungen können über zwei Wege entstehen:

- Outgoing: Ein shadow-Programm kann gewisse Informationen über sich selbst als eine Mitglied-Datei exportieren. Diese enthält alle Informationen, die nötig sind, um sich mit diesem Mitglied zu verbinden. Solche Dateien lassen sich dann an andere Server geben und dort importieren. Mit dem Importieren versucht shadow dann eine Verbindung mit dem anderen Mitglied aufzubauen.

- Incoming: Auf der anderen Seite muss die Verbindung entgegengenommen werden. Dafür hört jedes shadow-Programm konstant auf einem TCP-Port. Elixir macht diesen Prozess sehr einfach und erlaubt es, von einem *listening*-Prozess aus für jede neue Verbindung einen eigenen Prozess zu starten. Dafür wird lediglich eine rekursive Funktion verwendet, welche für jede neue Verbindung die Kontrolle über diese Verbindung an einen eigenen Prozess überträgt. Das Ganze ist mit zwei einfachen Funktionen umgesetzt:

```
def accept(port) do
  {:ok, listen} =
    :gen_tcp.listen(
      port,
      [:binary, packet: :line, active: true, reuseaddr: true]
    )
```

[14]Einführung in *Distributed Systems* mit Elixir, Jakob Klemm: https://orion.jeykey.net/distributed_systems.pdf, heruntergeladen am: 2.06.2020.

```
  loop(listen)
end
```

In der `loop/1` Funktion wird dann auf neue Verbindungen gewartet und mit den entstandenen Sockets der Router aufgerufen. Dieser kümmert sich dann um das Starten des eigentlichen Mitglied-Prozesses.

```
def loop(listen) do
  {:ok, socket} = :gen_tcp.accept(listen)
  pid = Shadow.Routing.incoming(socket)
  :gen_tcp.controlling_process(socket, pid)
  loop(listen)
end
```

Zwar haben diese beiden Arten Verbindungen aufzubauen wenig gemeinsam. Sobald die Verbindungen aber vollständig sind, verwenden sie die gleiche Modul-Struktur. Beide Arten werden in `Elixir` mit einem `GenServer` repräsentiert. `Elixir` verwendet, anders als andere Sprachen, ein System von modul-weiten Callbacks. Gewisse Funktionen im Modul können dann auf bestimmte Namen abgleichen und nur auf die richtigen Nachrichten antworten. Die `TCP` Funktionalität in `Elixir`, welche wiederum auf `Erlang` basiert, verwendet sowohl für ausgehende, als auch für eingehende `shadow`-Verbindungen die gleiche Struktur, weswegen die Unterscheidung der zwei Arten von Verbindungen nur während dem Aufbau einer neuen Verbindung wichtig ist. Danach werden alle Mitglieder gleich behandelt. Jedes Mitglied ist zusätzlich noch bei einem zentralen Prozess, dem Router, registriert. Dieser kümmert sich um das Zuweisen von Nachrichten und verwaltet die Verbindungen zu allen Mitglieder.

5.2.1 Routing

Der wichtigste zentrale Prozess in `shadow` nennt sich *Routing*. Er ist, wie der Name schon verrät, dafür verantwortlich, eingehende Nachrichten an den richtigen Mitglied-Prozess (ein Mitglied-Prozess ist dabei einfach die Repräsentation eines anderen Mitglieds in `Elixir`) zu senden. Gleichzeitig kümmert er sich auch um das Verwalten und Beobachten der Mitglied-Prozesse und agiert als Schnittstelle für eingehende Nachrichten und verschiedenste Funktionsaufrufe. Anders als Mitglied-Prozesse, welche für jedes Mitglied gestartet werden, kann jede `shadow` Instanz nur einen einzigen *Routing*-Prozess haben, weswegen der Name für die ganze Applikation registriert ist.

Anders als in den Mitglied-Prozessen werden im Router andere Informationen gespeichert. Anstelle des Sockets, der die tatsächliche Verbindung zu einem anderen Mitglied repräsentiert, werden im Router Referenzen gespeichert. Diese erlauben es, die einzelnen Mitglieds-Prozesse zu überwachen und sie gegebenenfalls aus der Routing-Tabelle zu entfernen. Damit garantiert der Router, dauerhaft die richtigen Informationen zu speichern und kann somit verhindern, dass Nachrichten an Mitglieder geschickt werden, die nicht mehr existieren.

Sobald eine Nachricht im Router landet, wobei nicht unterschieden wird, woher sie kommt, sucht der Router das passende Mitglied. Dafür wird die `XOR`-Funktion verwendet, deren Vorteile und Funktionsweise bereits beschrieben wurden. Der tatsächliche Code für die Funktion ist trotz ihrer Wichtigkeit äusserst einfach:

```
def distance(source, target) do
  source ^^^ target
end
```

Diese Funktion wird dann eingesetzt, um alle Einträge im Routing-Prozess zu sortieren, bis der nächste gefunden wird. Der Code dafür sieht wie folgt aus:

```
if Local.is_local?(message.target) do
  {:reply, :__SERVER__, state}
else
  distanced =
    Enum.map(state, fn {_k, v} ->
      Key.distance(message.target, v.key)
    end)

  min = Enum.min(distanced)

  member =
    Enum.find(state, fn {_k, v} ->
      min == Key.distance(message.target, v.key)
    end)

  {:reply, elem(member, 1), state}
end
```

Mit diesen zwei einfachen Funktionen kann der Prozess das nächste Mitglied für eine Nachricht finden. Hier sehen wir auch den Test, ob die Nachricht *local* ist. Dafür wird im Hintergrund `hunter` über eine Kommandozeile aufgerufen, um herauszufinden, ob der Empfänger einer Nachricht lokal registriert ist.

Ein Grossteil der wichtigen Funktionen und Module kümmern sich um die Verbindungen zwischen Mitgliedern und dem Senden von Nachrichten. Dazu gibt es aber noch eine Vielzahl von zusätzlichen Funktionen, selten aufgerufen, aber trotzdem wichtig für die Funktionsweise, so beispielsweise die Erstellung neuer `Kademlia`-Routing-Adressen:

```
Stream.repeatedly(fn -> :rand.uniform(255) end)
|> Enum.take(20)
|> :binary.list_to_bin()
|> :binary.decode_unsigned()
```

Neben dem Routing-Prozess existieren noch weitere Prozesse, welche die einfache Verwendung des Routing-Prozesses erst ermöglichen, darunter eine *Registry* für alle Mitglieds-Prozesse. Diese erlaubt die Prozesse direkt unter ihrer eindeutigen Identifizierung zu speichern, anstatt komplizierte `Elixir`-Prozess-ID's speichern zu müssen.

5.2.2 Portal

Sollte `hunter` im obigen Code-Stück gefunden haben, dass der Empfänger einer Nachricht lokal registriert ist, so muss die Nachricht an ihn, beziehungsweise an `hunter` übertragen werden. Dafür wird ein eigenes Modul namens `Portal` verwendet. Es benutzt einen `IPC`-Socket um mit `hunter` zu kommunizieren. Da `Elixir`, beziehungsweise `Erlang` für das entwickeln von Telecom und Nachrichtensystemen entwickelt wurden, funktioniert Vieles über sehr ähnliche Schnittstellen. Für einen `IPC`-Socket zum Beispiel verwendet man einen normalen `TCP`-Socket, gibt allerdings eine lokale Adresse an. Dies würde es theoretisch sogar erlauben, auch diese Schnittstelle über das gleiche Mitglied-Modul zu verwalten. Aus Gründen der Modularität wurde das Portal allerdings als eigenes Modul ausgelagert. Deshalb sieht der relevante Code, welcher sich in nur einer Funktion ausdrücken lässt, wie folgt aus:

```
def init(_params) do
  path = Local.path() <> "shadow.sock"
  :gen_tcp.connect({:local, path}, 0, [:binary])
end
```

Die eigentlichen Callbacks und Verarbeitungen funktionieren Identisch wie für eingehende `TCP`-Nachrichten.

5.3 Container

Sobald die Nachrichten ein Mitglied erreichen um dort verarbeitet zu werden, landen sie bei *Container*. Bevor aber die Funktionsweise dieses Komponenten erklärt werden kann, muss zuerst die potentiell verwirrende Namensgebung angesprochen werden:

- *Container* ist ein Programm, welches aktiv auf eingehende Nachrichten hört.
- Sobald eine Nachricht ankommt, wird eine Instanz der passenden Verarbeitungseinheit gestartet. Diese sind vergleichbar mit den angesprochenen `Cloud Functions`, lassen sich aber auch als kurzlebige Container beschreiben.

Container besteht aus zwei Komponenten:

- Der Verwaltungsteil ist für die Kommunikation zu `hunter` und dem restlichen System zuständig. Das heisst, dass das System in einem ersten Schritt die Daten von `hunter` erhält und kategorisiert. Wichtig ist anhand der Metadaten der Nachricht festzustellen, wie auf die Nachricht reagiert werden soll und wie die Daten verarbeitet werden sollen.
- Der Verarbeitungsteil ist das Herzstück von Container. Für jede eingehende Nachricht wird eine Instanz einer Verarbeitungseinheit gestartet. Diese enthält passenden `NET-Script` Code, welcher die Daten verarbeitet. *Container* fängt direkt an, den `NET-Script` Code auszuführen und verarbeitet die Daten. Dabei ist es in `NET-Script` möglich Daten weiterzusenden oder Modifikationen zum lokalen Speicher durchzuführen. Wenn *Container* seinen `NET-Script` Code fertig ausgeführt hat, beendet er die Verarbeitungseinheit und meldet sie beim Verwaltungsteil ab. Eine einzelne Recheneinheit für die Verarbeitung einer Nachricht existiert also nur so lange, wie die Nachricht selbst.

Dieser Ansatz hat mehrere Vorteile: Er abstrahiert das System so, dass die uninteressanten Teile, die niemals durch den Nutzer beeinflusst werden, wegfallen und sich Entwickler vollständig auf die Anwendungsfälle konzentrieren können. Dieses Abstrahieren hat aber auch den Vorteil, dass es deutlich

sicherer ist, denn NET-Script kann nur Daten verarbeiten und nimmt deshalb auch keinen Einfluss auf die grundlegende Systemarchitektur. Zusätzlich kümmert sich Container um verschiedene Funktionen, wie beispielsweise das Routing von Nachrichten zum richtigen Container, was normalerweise für Entwickler unnötigen Aufwand bedeutet.

5.4 NET-Script

`NET-Script` ist die Programmiersprache, in der die Logik für die einzelnen Container geschrieben ist. Die Syntax orientiert sich an der Sprache `elisp`, welche für die Konfiguration von Emacs genutzt wird. `elisp` gehört zur Familie der `Lisp`-Sprachen[15].

`Lisp` ist eine der ersten Sprachen, welche nicht kompiliert sondern interpretiert wurde. Das bedeutet, dass der Code nicht in ein für den Prozessor verständliches Format verwandelt wird, sondern, dass der Text während der Laufzeit des Programms interpretiert und übersetzt wird. Dies setzt aber voraus, dass es während der Laufzeit ein anderes Programm gibt, welches den Text übersetzen kann.

`Lisp` ist für *Container* geeignet, da die Sprache ziemlich einfach vom Textformat in ein für den Computer verständliches Format umgewandelt werden kann. Zudem ist es bei der richtigen Implementierung nicht schwer, neue Befehle einzuführen, ohne die aktuelle Struktur durcheinander zu bringe. Dies wird zusätzlich vereinfacht, da aus der Sicht des Übersetzers alles als eine Funktion angesehen werden kann.

Da die Struktur und die Syntax der Sprache nahe an `Lisp` sind, sollte der Lernaufwand sehr gering sein. Ein beispielhaftes Programm in `NET-Script` sieht wie folgt aus:

```
(defun main ()
  (let result (concat MESSAGE "!"))
  (route result RECEIVER))
```

Wenn ein neuer Container gestartet wird, wird eine besondere Funktion namens `main` ausgeführt. In der aktuellen Implementierung existieren die folgenden Tokens und Befehle:

[15] Wikipedia: Lisp & Lisp Ähnliche Programmiersprachen: `https://de.wikipedia.org/wiki/Lisp`, heruntergeladen am: 6.06.2020.

- `defun`: Dieser Token ist für die Definition einer neuen Funktion zuständig. Damit der Interpreter aber zufrieden ist, müssen nach dem Token noch mindestens drei weitere Tokens gegeben sein, nämlich der Name der Funktion, in diesem Fall `main` und eine öffnende und schliessende Klammer. In diesen lassen sich Funktionsargumente angeben. Die Namen der Parameter werden dann beim Aufruf der Funktion den tatsächlichen Werten zugewiesen.

- `let`: Dieser Token ist für die Erstellung von neuen Variablen zuständig. Diese Funktion verlangt zusätzlich noch zwei weitere Argumente. Zum einen muss der Name der Variable gegeben sein, zum anderen der Wert der ihr zugewiesen werden soll. Dieser kann allerdings auch aus Funktionsaufrufen oder anderen Tokens bestehen. Im obigen Beispiel ist das letzte Argument das Resultat, welches aus dem `concat` Befehl kommt.

- `concat`: Dieser Befehl nimmt die beiden Zeichenketten und hängt sie aneinander. Anschliessend wird die neu entstandene Zeichenkette als Rückgabewert der Funktion angesehen und im obigen Beispiel in die Variable `result` geschrieben. Die Variable `MESSAGE` ist eine vordefinierte Variable und enthält den Inhalt der erhaltenen Nachricht von Shadow. Diese und ähnliche Funktionen zum verarbeiten und Manipulieren existieren speziell für das System und sind für das verarbeiten von `Engine: Orion`-Nachrichten ausgelegt.

- `route`: Dies ist einer der wichtigsten Funktionen von `NET-Script` und erlaubt es, Nachrichten aus dem Container zu anderen `Engine: Orion` Komponenten zu schicken. Diese Funktion nimmt zwei Parameter entgegen: als Erstes die Nachricht, die gesendet werden soll, und als Zweites die Adresse des Ziels, wobei es keine Rolle spielt, ob diese lokal existiert oder nur auf einem anderen Server zu finden ist.

Zwar ist die aktuelle Implementierung von `NET-Script` noch stark limitiert, trotzdem lassen sich bereits die stärken eines stark integrierten Systems erkennen.

6 Auswertung

Nachdem nun `Engine: Orion` sowohl technisch, als auch praktisch erklärt wurde, müssen der tatsächliche Nutzen und die Einsatzmöglichkeiten

besprochen werden. Als Allererstes ist es aber wichtig, nochmals an die ursprünglichen Ziele zu erinnern:

- Eine technische Revolution in der Art, wie wir miteinander kommunizieren ist von Nöten.
- `Engine: Orion` soll diesen Fortschritt nicht alleine umsetzen, sondern mit einem Prototypen zeigen, was tatsächlich möglich ist, sodass andere daraus sowohl kommerzielle, als auch freie Produkte und Dienste bauen können.

Erfüllt die hier beschriebene Umsetzung diese Ziele?

- Es ist schwer ein vergleichbares System zu finden, welches den gleichen theoretischen Komfort bei der Entwicklung von verteilten Kommunikationsprogrammen anbietet.
- Das aktuelle System löst oder umgeht viele der zentralen Probleme.
- Normale Nutzer können Programme basierend auf `Engine: Orion` nutzen, ohne sich Gedanken über die technische Umsetzung zu machen.

Zwar wirkt dies äusserst positiv und erfolgreich, allerdings gibt es einige wichtige Probleme:

- Das aktuelle System ist nicht sonderlich praktisch. Der Aufwand einer eigenen Konfiguration mit mehreren Servern und Eingabe- / Ausgabe-Programmen ist technisch äusserst komplex.
- Durch verschiedene Sprachen, Komponenten und Geräte, sowie die fehlende Optimierung ist die aktuelle Umsetzung nicht schnell. Zwar reicht es für einfache Programme wie einen Chat, allerdings wäre es zu langsam für Videospiele, Echtzeit-Applikationen, etc.

Zwar ist `Engine: Orion` im aktuellen Zustand noch lange davon entfernt, tatsächlich zuverlässig Daten zu verarbeiten oder Nutzer zu beliefern, trotzdem ist es ein funktionsfähiges System, welches gewisse Schritte der aktuellen Entwicklung von verteilten Nachrichtensystemen unnötig macht oder stark vereinfacht.
Eigentlich hätten noch eine numerische Auswertung der Geschwindigkeit und Zuverlässigkeit gemacht werden sollen, mit welcher die obigen Aussagen statistisch belegt werden könnten, allerdings ist es unglaublich schwer, ein solch komplexes System zuverlässig zu testen. Neben den verschiedensten potentiellen Fehlerquellen im System selbst, gibt es Hunderte von möglichen Engpässen bei zusätzlichen Programmen und Protokollen im System, beispielsweise das Netzwerk oder das Betriebssystem der einzelnen Geräte.

7 Ausblick

Was auch immer mit diesem Projekt noch geschehen mag, Potential hat es auf jeden Fall genug. Die möglichen Verbesserungen, Erweiterungen und Optimierungen sind nahezu unbegrenzt. In den nächsten Abschnitten sollen einige der nächsten Schritte besprochen werden, wobei die Realisierbarkeit immer im Blick behalten werden muss.

Dazu ist es wichtig, die Ziele und Prinzipien hinter `Engine: Orion` im Kopf zu behalten: Während andere versuchen, kommerzielle Lösungen zu entwickeln und diese für möglichst viele Nutzer möglichst einfach anzubieten, soll mit `Engine: Orion` etwas anderes erreicht werden. Verschiedene Probleme und nötige Veränderungen wurden bereits angesprochen. Anstatt diesen Wandel aber alleine zu bringen, versucht `Engine: Orion` ein Schritt in einer langen, technischen Revolution hin zu grundlegend neuer Infrastruktur zu sein. Mit dem Bewusstsein für dieses Ziel sollen nun die nächsten Schritte behandelt und später auch umgesetzt werden.

7.1 Vervollständigung

Als erster Schritt, bevor Visionen oder Träume umgesetzt werden können, muss den ursprünglichen Zielen nachgekommen werden. Auch wenn `Engine: Orion` von Anfang an als Prototyp gehandhabt wurde, liegt aktuell mehr der Prototyp eines Prototypen vor. Zwar erfüllt dieser seine Aufgaben, kann aber darüber hinaus eher wenig. Neben der offensichtlich fehlenden Nutzerfreundlichkeit ist das System als Ganzes auch noch nicht allgemein einsetzbar, denn viele der Funktionen sind noch zu spezifisch auf die Demo sowie die Entwicklung ausgelegt. Zwar existieren die Grundlagen und sie funktionieren auch relativ zuverlässig, es fehlt aber noch an verschiedenen praktischen Funktionen um das System zu vervollständigen. Natürlich werden zusätzliche Funktionen und zusätzliche Zuverlässigkeit benötigt, um das System grossflächig einsetzbar zu machen. Allerdings ist die tatsächliche Skalierbarkeit nicht definitiv geklärt.

Beispielsweise gibt es technische Limitierungen mit dem aktuellen `TCP`-Protokoll. Für eine zukünftige *globale* Umsetzung müsste wahrscheinlich `UDP` oder das neue `QUIC`-Protokoll verwendet werden. Allerdings sind Fragen wie diese sehr schwer zu beantworten, da es unglaublich komplex ist, verteilte Systeme zu testen. Denn die tatsächlichen Bedingungen eines verteilen Systems über Hunderte von *Servern* lassen sich kaum simulieren, ohne tatsächlich Hun-

derte von Instanzen auf hunderten von physikalisch getrennten Geräten zu starten, was aus praktischen und finanziellen Gründen natürlich kaum möglich ist. Trotzdem lässt sich mit einer besseren Integration und dem Wechsel auf schnellere Programmiersprachen und bessere Protokolle, sowie der Optimierung des Codes und des Datenflusses noch viel Leistung gewinnen.

7.2 Container

Der aktuelle Entwicklungsstand ist sowohl bei den Container als auch bei der Programmiersprache und Entwicklungsumgebung noch in den Kinderschuhen. Auch wenn mittlerweile die wichtigsten Funktionen umgesetzt wurden, gibt es immer noch viele Leistungseinbussen und strukturelle Probleme im Code sowie im Datenfluss.

Neben der Vervollständigung, mit der vor allem `NET-Script` zu einer vollständigen, *turing-fähigen* Programmiersprache gemacht werden soll, müssen auch gewisse Komponenten im restlichen System optimiert und verbessert werden. Zwar ist `NET-Script` in der Lage, einfache Befehle auszuführen und Nachrichten grundsätzlich zu verarbeiten, allerdings ist dies aktuell nur in gewissem Rahmen möglich. Auch wenn `NET-Script` nie ausserhalb dieser Umgebung eingesetzt werden sollte, fehlen aktuell doch kritische Funktionen, welche zwingend benötigt werden, um mehr als einfache Demos für `Engine: Orion` zu schreiben.

7.3 Datenbank

Zwar existiert aktuell bereits eine Möglichkeit, Daten über die Verarbeitungszeit einer Nachricht hinaus zu speichern, allerdings wird zusätzlich noch eine klassische Datenbank für grössere Datenmengen benötigt, da der aktuelle Speicher, welcher bestehen bleiben soll, nur für kleinere, temporäre Werte gedacht ist. Natürlich müsste keine komplette Datenbank neu implementiert werden. Eine einfache Zugangsmöglichkeit direkt in `NET-Script` auf eine existierende Datenbank, sowie die Integration dieser in den *Container* oder `hunter` wäre bereits genug, um das System als Ganzes tatsächlich vielseitig einsetzbar zu machen.

7.4 Adressen

Für einen Prototypen mag es ausreichen, die Menge an verfügbaren Adressen zu limitieren und einen fixen Wert zu nutzen, beziehungsweise diese von

Hand zu verwalten. Für ein tatsächlich skalierbares, dynamisches System wird es aber wichtig, die Abschnitte von reservierten Adressen dynamisch zu machen. Dafür soll ein Prozess namens `Keyspace negotiations` eingesetzt werden, der es verbundenen Mitgliedern erlauben soll, über die Grösse der reservierten Abschnitte zu verhandeln.

Sobald sich aber die Grössen der Abschnitte während der Lebenszeit eines Objekts in einem dieser Abschnitte ändern, kommen neue Probleme auf:

- Externe Prozesse, die auf das Objekt zugreifen wollen, müssen über die Änderung informiert werden.
- Sollte dies nicht möglich sein (da die Prozesse beispielsweise kurzzeitig nicht verfügbar sind), muss das ursprünglich verantwortliche Mitglied in der Lage sein, eingehende Nachrichten für das Objekt an das neu verantwortliche Mitglied zu schicken.
- Da sich die Adresse aufgrund der obigen Probleme nicht ändern darf, muss sie beim neuen Mitglied reserviert bleiben. Dies bedeutet, dass die Veränderung der Abschnitte keine Vorteile bringen würde, da je nach Situation eine Vielzahl der neuen Adressen reserviert sein müsste.

Um diese Probleme zu umgehen, muss neben dem Adressen-System noch ein zusätzliches ID-System eingeführt werden, welches bereits teilweise im aktuellen System vorhanden ist. Mit diesem zweiteiligen System ist es möglich, nur die ID's zu reservieren, während die Adressen frei werden. Zwar muss damit jeder, der eindeutig mit einem Objekt kommunizieren will, einen zusätzlichen Wert speichern, aber es vereinfacht sowohl das verteilte Routing als auch das lokale.

7.5 DNS

Das eben angesprochene Zwei-Komponenten-System geht klar gegen die Nutzerfreundlichkeit. Daher ist es vorteilhaft, ein zusätzliches System einzuführen, das bei diesem Werte-Paar eine einheitliche Identifizierung über einen einzigen Wert erlaubt.

Da mit einem solchen System kein technisches, sondern eher ein praktisches Problem gelöst werden soll, stehen viel mehr Möglichkeiten zur Verfügung:

- In seiner einfachsten Form präsentiert ein Namens-System die beiden Werte einfach als einheitlichen String, der dann in allen Systemen ver-

wendet werden kann. Technisch wäre diese Variante ganz klar die Einfachste, bringt allerdings eher wenig Nutzerfreundlichkeit, da die neuen Werte zufällig wirken würden.

- Tatsächlich gibt es bereits ein dezentralisiertes Domain-System, nämlich im `Dark-Net` in Form von `=Onion-Adressen`. Mit diesen Adressen ist es möglich, einzelne Mitglieder im System genau zu finden. Da die Adressen direkt von den privaten und öffentlichen Schlüsseln abgeleitet werden, gibt es Programme und Scripts, die es erlauben, Adressen gezielt zu generieren. Der limitierende Faktor dabei ist die verfügbare Rechenleistung, denn das passende Schlüsselpaar wird durch Ausprobieren gefunden. Damit ist es Nutzern möglich, zumindest die ersten Zeichen einer solchen Adresse zu bestimmen, auch wenn der Rest weiterhin zufällig ist.

- Neben den *deterministischen* Varianten, die durch Kryptographie und Mathematik die Adressen finden und garantieren, gibt es noch einen anderen Weg. Ähnlich wie das aktuelle Domain-Name-System wäre ein System denkbar, das Adressen per Broadcast an möglichst viele Mitglieder verteilt, die dann ihren eigenen Routing-Table dementsprechend anpassen. Natürlich wird irgendein System benötigt, um die Integrität der Adressen zu validieren, dafür lässt sich aber potentiell direkt ein anderer *nächster Schritt*, nämlich ein Währungssystem, verwenden. Auch dieses System garantiert nicht, dass Nutzer tatsächlich das richtige Mitglied finden. Je nach Implementierung könnte es sogar passieren, dass mehrere Mitglieder in einem Netzwerk die gleiche Adresse anbieten. Natürlich führt dies wieder zu neuen Gefahren und Problemen, könnte aber den Wechsel für neue Nutzer einfacher machen, da es sehr nahe am aktuellen `DNS`-System ist.

7.6 Balance

In einem System ohne klare Rollen oder genauer: in einem System, in dem jedes Mitglied gleichwertig ist und alle möglichen Rollen gleichzeitig einnimmt, braucht es einen Mechanismus, der Nutzer belohnt, die mehr Arbeit übernehmen und mehr für das Netzwerk als Ganzes beitragen.

Grundsätzlich gibt es zwei Möglichkeiten, um solche unbelohnte Arbeit zu vermeiden:

- Man kann dafür sorgen, dass jedes Mitglied ungefähr gleichmässig ausgelastet ist. Dafür muss beim Weiterleiten jeder Nachricht in Betracht

gezogen werden, wie viel ein bestimmtes Mitglied in letzter Zeit ausgelastet war, um dementsprechend den Weg der Nachricht anzupassen. Dies führt zwar dazu, dass jedes Mitglied mehr oder weniger gleich viel oder zumindest proportional gleich viel Daten verarbeitet, aber es führt zu einem weniger effizienten System, da Nachrichten in bestimmten Situationen nicht den direkten Weg nehmen.

- Mit einem Währungs- oder Punktesystem wäre es möglich, Mitglieder direkt dafür zu belohnen, wenn sie mehr Arbeit übernehmen. Innerhalb dieser Idee gibt es noch zwei Unterkategorien, die unterschieden werden müssen:
  - Ein lokales Punktesystem, welches jedem Mitglied im Routingtable eines einzelnen Mitglieds einen Punktestand zuweist. Der Punktestand wird mit eingehenden Nachrichten angepasst, zieht aber auch Laufzeit und Netzwerk-Status in Betracht.
  - Ein globales Währungssystem, vergleichbar mit einer Cryptocurrency. Das erfolgreiche Verarbeiten und Weiterleiten von Nachrichten würde in einem solchen System also mit Guthaben belohnt. Hierbei müsste man entscheiden, ob der Vorrat an dieser Währung limitiert ist, die Belohnung also durch die Mitglieder ausgezahlt wird, an die die Nachricht ausgeliefert werden soll, oder ob der Vorrat ähnlich wie bei `Bitcoins` unendlich (oder nahezu unendlich) ist, und für jede Nachricht, für jeden Eintrag neues *Geld* gedruckt wird. Da nicht alle Mitglieder in einem verteilten Nachrichten und Datensystem gleich viel Last auf das System bringen, würde eine solche Währung einen Grund schaffen, dem System zu helfen und Nachrichten zu verarbeiten.

KANTI BADEN

Bestätigung

Ich/Wir erkläre/-n hiermit, dass meine/unsere Maturaarbeit von mir/uns verfasst oder entwickelt und nicht als Ganzes oder in Teilen kopiert wurde.

Aus Quellen übernommene Teile sind – nach den entsprechenden Regeln – als Zitate erkennbar gemacht. Alle Informationsquellen sind in einem Literaturverzeichnis aufgeführt.

Vorname/-n, Name/-n:	Abteilung/-en:
Dominik Keller, Jakob Klemm	G3a

Maturaarbeit:

Engine: Orion

Ort, Datum:

4.06.2021, Baden Schweiz

Unterschrift/-en:

D. Keller Jakob Klemm

Eine Kopie der Bestätigung geben Sie – mit Originalunterschrift versehen – bei der Schlusspräsentation im November ab.

Kantonsschule Baden
5400 Baden
www.kanti-baden.ch

KANTON AARGAU