Algorithmen für verteilte Systeme in unbemannten Luftfahrzeugen

Artsiom Kaliaha

BACHELORARBEIT

eingereicht am

Fachhochschul-Bachelorstudiengang

Automotive Computing

in Hagenberg

im Februar 2021

Betreuung:

Mag. Dipl.-Ing. Dr. Andreas Müller B.Sc.

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt. Die vorliegende, gedruckte Arbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Hagenberg, am 1. Februar 2021

Artsiom Kaliaha

Inhaltsverzeichnis

Kurzfassung

Drohnen ändern den Alltag auf überraschende Weise: sie retten in Katastrophengebieten, sichern die medizinische Versorgung in Konfliktzonen, reduzieren Verkehr und Emissionen und machen riskante Berufe sicherer. Heutzutage werden Drohnen vielseitig eingesetzt. Andere Gebiete, bei denen Drohnen zum Einsatz kommen, sind das Militär (Steuerung von Drohnenschwärmen), der Telekommunikationsbereich (schneller Aufbau von Netzwerken auf Nachfrage) und die Ausbildung (Aufnahme von 360-Grad-Videos).

Es gibt mehrere Forschungsrichtungen, die Funktionsweise einzelner Drohnen und Drohnenschwärmen optimieren. Eine von dennen ist Swarming. Swarming ist ein Bereich von Multiagentensystem, der Drohnen ermöglicht, bestimmte Tierarten nachzuahmen. Die Schwarmintelligenz ermöglicht es hunderten, wenn nicht tausenden von Drohnen, zusammenzuarbeiten, um herausfordernde Aufgaben zu erledigen. Heutzutage forscht die gesamte Robotikindustrie auf dem Gebiet der kollaborativen Robotik, bei der Roboter nebeneinander arbeiten und von Menschen trainiert werden. Schon in Naher Zukunft werden wir aber in der Zeit der Cloudrobotik leben, in der Roboter ohne menschlicher Interaktion als ein einziger Organismus agieren und denken werden. Fortschritte in den Bereichen der Künstlicher Intelligenz und der Cloudrobotik treiben die Schwarmtechnologie an und werden dazu beitragen, dass Drohnen nicht nur mit ihren Operatoren, sondern auch miteinander kommunizieren werden. Heutzutage wird jede Drohne von einem oder mehreren Menschen gesteuert. Die Drohnen von morgen werden möglicherweise überhaupt keine Operatoren brauchen. Forscher untersuchen Möglichkeiten, Menschen durch Künstliche Intelligenz und Algorithmen für verteilte Systeme zu ersetzen. Da eingebettete Systeme mit jedem Jahr leistungsstärker werden, haben sie genug Kapazitäten, um Sensordaten ohne Cloudverbindung zu bearbeiten, Rechenoperationen gleichmäßig untereinander aufzuteilen und Drohnen mit speziellen Funktionen im Schwarm redundant abzusichern.

Alle diesen Funktionen brauchen klassische Algorithmen für verteilte Systeme, die schon seit Jahren bei Cloudlösungen zum Einsatz kommen. Sie beruhen auf den Algorithmen aus den 80ern und 90ern, die immer noch in Produktivsystemen verwendet werden und als Inspiration für neue und optimalere Algorithmen dienen. Der Fokus dieser Bachelorarbeit liegt auf der Untersuchung verfügbarer Algorithmen für verteilte Systeme und auf der Recherche nach einem optimalen Concurrency Modell für robuste und fehlertolerante eingebettete Systeme.

Abstract

Drones change everyday life in surprising ways: they save in disaster areas, provide medicine in conflict zones, reduce traffic flow and emissions, and make risky jobs safer. Today, drones are used in many ways. Other areas where drones are used are the army (control of swarms of drones), the telecommunications sector (on demand network provisioning) and training (360-degree videos).

There are several research areas that focus on optimizing individual drones operation and swarms of drones control. One of these research topics is swarming. Swarming is a subfield of multi-agent systems that allows drones to mimic behavior of certain animal species. Swarm intelligence allows hundreds, if not thousands, of drones to work together to complete challenging tasks. Today the entire robotics industry is doing research in the field of collaborative robotics, which enables robots to work side by side with humans and to be trained by humans. In the near future, however, we will be living in the age of cloud robotics, in which robots will act and think as a single organism without human interaction. Advances in artificial intelligence and cloud robotics are driving swarm technology and will help drones communicate not only with their operators, but also with each other. Nowadays, each drone is controlled by one or more people. Tomorrow drones may not need operators at all. Researchers are investigating possibilities of replacing humans with artificial intelligence and algorithms for distributed systems. As embedded systems become more powerful each year, they get more and more capacity to process sensor data without cloud connection. Innovative approaches to building swarm systems suggest splitting computation evenly across all drones and redundantly securing drones which fulfill special functions in a swarm or carry unique hardware.

All of these functions require classic algorithms for distributed systems that were used in cloud solutions for years. They are based on the algorithms from the 80s and 90s, which are still used in production systems and serve as inspiration for more optimal algorithms. The goal of this bachelor thesis is to investigate available algorithms for distributed systems and find an optimal concurrency model for robust and fault-tolerant embedded drone systems.

Kapitel 1

Einleitung

Wir leben in der Zeit Künstlicher Intelligenz, Big Data und intelligenter Robotersysteme. Heutzutage ist die Vielfalt der Robotersysteme sehr beeindruckend: Roboter finden Einsatz in Form autonomer Autos, intelligenter Produktionstechnik und auch *Drohnen*. Vor 5 Jahren waren Drohnen noch manuell gesteuert, und erlaubten keine anspruchsvolle Rechenoperationen direkt auf der Hardware. Zukünftig sollen Drohnen leistungsfähiger sein und noch autonomer funktionieren und das nicht nur einzeln sondern auch in Schwärmen. Der Fokus dieser Arbeit liegt im Gebiet der Drohnenschwärme, genauer gesagt in der Betrachtung dieser als *verteiltes System*. Dadurch, dass Drohnen miteinander kommunizieren, können sie in einem Schwarm voll autonom und ohne Kollisionen fliegen. Die Kommunikation im Schwarm liefert in diesem Fall zusätzliche Informationen für die Funktion des autonomen Fliegens. Die Anwendung von Algorithmen für verteilte Systeme macht diese Bachelorarbeit auch dadurch interessant, dass diese Algorithmen mehr Anwendungsfälle für Interaktionen innerhalb eines Schwarmes und auch zwischen einem Schwarm und der Cloud bieten. Ein weiterer möglicher Anwendungsfall ist die Entlastung von Operatoren, die diese Drohnen normalerweise steuern. Die Entwicklung von Drohnen mit einem hohen Grad an Autonomie bietet den Operatoren im Laufe einer Mission bzw. eines Auftrages mehr Freiheit, andere nützliche Aufgaben zu erfüllen (Kamera ausrichten, Objekt beobachten, 3D Mapping des Geländes steuern, usw).

Fortschritte in Kommunikationstechnologien (vor allem 5G), Künstlicher Intelligenz und eingebetteter Hardware haben die zweite Welle von *Drohnentechnologie* ausgelöst. Manche Anwendungsfälle, die früher nicht denkbar waren, werden derzeit in Forschungsinstituten und Universitäten (in Europa vor allem EPFL [1] und ETH Zürich) erforscht. Innovative Startups versuchen, all diesen Anwendungsfälle auf den Markt zu bringen. Die Drohnenindustrie bietet bahnbrechende Anwendungsfälle und dadurch auch neue Herausforderungen.

Das Problem, das diese Arbeit löst befasst sich mit einer solchen Herausforderung, und zwar *der Synchronisation zwischen mehreren Knoten in einem Cluster von Drohnen*. Wenn Drohnen eine abstrakte Kommunikationstechnologie verwenden, muss ein Algo-

[1] Die École polytechnique fédérale de Lausanne ist eine technisch-naturwissenschaftliche Universität in Lausanne, Schweiz.

rithmus gewählt werden, der Daten von Drohnen in einem Schwarm zuverlässig synchronisiert. Um eine passende Lösung dafür zu finden, wurden bestimmte Algorithmen verglichen. Zwei Algorithmen, die sich bei der Betrachtung verteilter Systeme und der Synchronisation von Konten besonders auszeichnen, sind *Paxos* und *Raft*. Das Problem der Synchronisation zwischen den Knoten eines Systems wird in der Informatik auch *„Problematik der Replikation des Zustandsautomaten“* genannt. Langjährige Erfahrungen der Cloudprovider, die diese Algorithmen für mehrere Bereiche ihrer Infrastruktur einsetzen (vor allem Speicherdienste und *MapReduce* [2] Clusters), haben die Nutzbarkeit dieser Algorithmen bewiesen. Zwar werden beide Algorithmen in zahlreichen Produktivsystemen benötigt, die Meinungen über die jeweiligen Stärken und Schwächen der Algorithmen sind jedoch noch gespalten.

In einem engen Zusammenhang mit verteilten Systemen stehen *Concurrency Modelle*. Ergänzend zu den unabhängigen Fehlern der Teilkomponenten und dem Fehlen der Global Clock ist die Concurrency eine der wichtigsten Eigenschaften der verteilten Systeme. In dieser Bachelorarbeit werden die Eigenschaften mehrerer Concurrency Modelle verglichen, um ein dafür am besten geeignetes Modell für die Entwicklung verteilter Systeme zu finden.

Überblick über das Dokument: im 2. Kapitel 2 wird das Thema der Drohnen und der Interaktion von Drohnen in einem Schwarm betrachtet: was sind die Vor- und Nachteile bei der Verwendung eines Schwarms, welche Schwierigkeiten bringt die Steuerung eines Schwarms mit sich, welche theoretischen Frameworks gibt es für die Steuerung verteilter Agenten. Außerdem wird das Thema der verteilten Systeme behzandelt: theoretische Grundlagen werden erläutert und Algorithmen für verteilte Systeme verglichen. Anschließend werden Concurrency Modelle für die Implementierung verteilter und ausfallsicherer Systeme untersucht. Das letzte Kapitel der Arbeit 3 befasst sich mit der Umsetzung eines der besprochenen Algorithmen mit einem der ausgewählten Concurrency Modelle. In diesem Kapitel wird der ausgewählte Algorithmus im Detail analysiert und erklärt. Welche Entscheidungen wurden bei der Umsetzung getroffen und weshalb. Zudem zeigt dieses Kapitel, welche Vorteile die ausgewählten Werkzeuge bieten.

[2] MapReduce ist ein Programmiermodell für nebenläufige Berechnungen über große Datenmengen auf Computerclustern.

Kapitel 2

Analyse

Um die Problemstellung der Arbeit besser zu verstehen, muss man einige grundlegende Konzepte der *Schwarmrobotik* und verteilter Systeme verstehen. Außerdem gibt dieses Kapitel einen Überblick auf den aktuellen Stand der Technik, Optionen und Empfehlungen für technische Umsetzung.

2.1 Vorteile von Drohnenschwärmen

Heutzutage werden Drohnen entweder einzeln verwendet oder sie finden in einem Drohnenschwarm ihre Anwendung. Beide Verwendungsarten haben Vor- und Nachteile und umfassende Anwendungsfälle. Um den Weg von einer Drohne zu einem Drohnenschwarm zu verstehen, muss man zuerst die Forschungsrichtungen im Feld von UAVs (Unmanned Aerial Vehicles) betrachten. Zu UAVs gibt es vier Forschungsschwerpunkte, die an entsprechende Herausforderungen gebunden sind:

1. Ein einziges UAV muss stets in der Lage sein, bestimmte Gebiete und Flächen zu überfliegen.
2. Objekte und Ereignisse von Interesse müssen von dem UAV erkannt werden können.
3. Um einen Synergieeffekt zu erreichen, ist die Kommunikation zwischen mehreren Drohnen essenziell.
4. Damit das Arbeiten der Drohnen an einem gemeinsamen Ziel ermöglicht wird, gilt es die Koordination eines Netzwerks von UAVs zu ermitteln.

Diese Bachelorarbeit fokussiert sich auf Drohnenschwärme und auf die Koordination zwischen den einzelnen Drohnen. Betrachtet man eine Gruppe von Drohnen, stellt man fest, dass man einen Algorithmus benötigt, der die Funktionsweise dieser Gruppe optimiert, damit diese durch Zusammenarbeit ein gemeinsames Ziel erreicht. Unter anderem können folgende Probleme durch die Anwendung eine Schwarms gelöst werden:

Verteilung von Rechenoperationen (optimale Nutzung von Ressourcen). Das ist ein klassischer Anwendungsfall aus der Informatik. Keine einzelne Drohne hat

genug Rechenleistung, um eine schwierige Operation im Stand-Alone-Betrieb bzw. ohne Verbindung zur Cloud zu berechnen. Dadurch, dass es in einem Schwarm mehrere Drohnen gibt, kann die Arbeit aufgeteilt werden. So einen Ansatz kann man "Ad-Hoc Cloud" nennen. Unter der optimalen Nutzung von Ressourcen kann man auch den Austausch von lokalen Karten und Sensorwerten verstehen.

Verteilung von Aufgaben. Für eine bestimmte Art von Problemen, die Schwarmintelligenz zur Lösung benötigt, ist es wünschenswert, dass jede Drohne eine bestimmte Funktion übernimmt und durch ihre speziellen Fähigkeiten (künstliche Intelligenz) oder Ausrüstung (Sensoren und Aktuatoren) hilft, das gemeinsame Ziel zu erreichen. Aus der Sicht der Ausfallsicherheit, Robustheit und Zuverlässigkeit eines verteilten Systems ist es immer besser, Tasks auf mehrere Geräte zu verteilen. Dieser Zusammenhang wurde schon durch die Erfahrungen von Google mit verteilten Systemen bewiesen. Andererseits muss man mit dem Kommunikationsoverhead und der gestiegenen Komplexität bei der Entwicklung rechnen.

Ausfallsicherheit. Ein Drohnenschwarm ist robust gegen Teilausfälle und bietet mehr Flexibilität bei Missionen. Wie oben schon erwähnt, können Aufgabenbereiche auf einzelne Drohnen und Drohnengruppen aufgeteilt werden.

Kollektive Entscheidungsfindung. Jede Drohne im Schwarm kann bzw. darf nur einen Teil vom ganzen Problem kennen. Für die Entscheidungsfindung braucht ein automatisiertes System sehr häufig nicht nur das lokale Wissen, sondern auch die Vogelperspektive oder den "Meinungsaustausch" mit anderen Drohnen.

2.2 Autonomie im Drohnenschwarm

Die im vorherigen Subkapitel erwähnten Vorteile eines Schwarms werden von Softwarekomponenten einer Drohne definiert, da diese Softwarekomponenten das Kommunikationsmuster bestimmen. Ein wichtiger Aspekt dabei ist die Autonomie von Drohnen. Autonomie ermöglicht einzelne Drohnen eines Schwarms lokale Entscheidungen zu treffen.

2.2.1 Methoden zur Kommunikation im Drohnenschwarm

Um die Vorteile, die Autonomie einzelner Drohnen mit sich bringt, ausnutzen zu können, können folgende systematische Ansätze angewendet werden:

Multiagentensystem. Ein Multiagentensystem ist eine Klasse von Algorithmen, in der einzelne Agenten basierend auf vordefinierten Regeln und Einschränkungen miteinander interagieren. Dadurch wird ein kollektives Verhalten ermöglicht.

Die Interaktionen im System finden sowohl zwischen Agenten untereinander als auch zwischen Agenten und der Umgebung statt. Dabei spielt die sogenannte *„Reward Function"* eine wichtige Rolle. Diese beschreibt, wie sich ein Agent verhalten muss. Der *„normative Inhalt"* schreibt dem Agenten vor, wie er bestimmte Aufgaben lösen soll. Im Multiagentensystem kennt ein Agent den gesamten Problembereich bzw. -raum nicht und muss die Lösung deswegen durch Lernen herausfinden. Multiagentensysteme üben Selbstorganisation, komplexe Verhaltensweisen

und auch Kontrollparadigmen aus, obwohl individuelle Strategien von allen Agenten einfach sind.

Ein Agent im Multiagentensystem hat einige wichtige Eigenschaften: *Autonomie* (ein Agent ist teilweise unabhängig, selbstbewusst und autonom), *lokaler Überblick* (kein Agent hat einen globalen Überblick und das System ist für einen einzelnen Agent zu komplex) und *Dezentralisierung* (kein Agent wird als Leader bezeichnet / gewählt).

Verteilte Problemlösung. Kooperative verteilte Problemlösung ist ein Netzwerk von halbautonomen Verarbeitungsknoten, die zusammenarbeiten, um ein Problem zu lösen. Dabei geht es um die Untersuchung der Problemaufteilung, Unterproblemaufteilung, Synthese vom Ereignis, Optimierung des Problemlösers und Koordination. Es ist eng mit der verteilten Programmierung und der verteilten Optimierung verbunden.

In der verteilten Problemlösung arbeiten mehrere Agenten daran, ein spezifisches Problem zu lösen. Das wichtigste in diesen Systemen ist, dass Kooperation erforderlich ist, weil kein einziger Agent genug Informationen, Wissen und Fähigkeiten hat, um das Problem zu lösen. Die eigentliche Herausforderung besteht in der Sicherstellung, dass die Informationen so aufgeteilt werden, dass die Agenten einander ergänzen und nicht miteinander im Konflikt stehen. Ein Algorithmus für die verteilte Problemlösung muss ein größeres Problem in Teilaufgaben unter Berücksichtigung der räumlichen, zeitlichen oder funktionalen Aspekte gliedern.

Ein Algorithmus für verteilte Problemlösung muss unter folgenden Einschränkungen handeln:

1. Kein Knoten hat genug Informationen, um das Problem selbstständig zu lösen.
2. Im System gibt es weder eine globale Steuerung, noch einen Speicher. Steuerung und Speicher sind verteilt.
3. Die Berechnung von Operationen auf einer lokalen CPU ist grundsätzlich schneller und weniger aufwändig als die Aufteilung derselben Operationen auf verteilte Systeme.
4. Darüber hinaus muss das zu lösende Problem modular sein. Außerdem darf es keinen einzigartigen Knoten geben, da dies zu einem *Bottleneck* im System führen kann. Ein solcher *einzigartiger Knoten* wäre jener, der im Cluster oder im Schwarm eine spezielle Rolle erfüllt. Eine solche Aufgabe könnte beispielsweise die eines Leaders oder eines Koordinators sein.

Schwarmintelligenz / Schwarmrobotik. Schwarmintelligenz ist das kollektive Verhalten von dezentralisierten, selbstorganisierten, natürlichen oder künstlichen Systemen. Das Konzept wird sehr oft im Bereich der Künstlichen Intelligenz eingesetzt.

Systeme mit Schwarmintelligenz bestehen typischerweise aus einer Population von Agenten und Boids, die untereinander und mit der Umgebung kommunizieren. Inspiration für solche Systeme kommt aus der Natur, vor allem aus biologischen Systemen. Die Agenten folgen sehr einfache Regeln. Obwohl es keine zentralisierte Steuerung gibt, führen lokale und bis zu einem gewissen Grad zufällige Interak-

tionen zur Entstehung eines intelligenten und globalen Verhaltens. Ein klassisches Beispiel aus der Natur sind Ameisenkolonien.

An dieser Stelle sieht man, dass die Grenzen zwischen Multiagentensystem, verteilter Problemlösung und Schwarmintelligenz verschwimmen können. In der Tat ist das Bilden einer klaren Grenze sehr schwierig.

2.2.2 Einblick in zukünftige Systeme

Moderne Systeme sind meist mit der Cloud verbunden oder brauchen zumindest eine kurze Synchronisation mit einem Cloud-Server. Nehmen wir an, dass eine Drohne eine Mission erfüllen soll, was sowohl Kommunikation zwischen Drohnen als auch zwischen dem Schwarm und der Cloud erfordert. Alle eingebetteten Systeme sind bis zu einem gewissen Grad in ihren Ressourcen (Rechenleistung, Akku, Konnektivität, Speicher) eingeschränkt. Dennoch wäre es sinnvoll, *Missionsinformationen* zwischen der Cloud und einer einzelnen Drohne (dem sogenannten Leader) zu synchronisieren und mittels eines Algorithmus im Schwarm zu verteilen.

Dies widerspricht dem Regel, dass jeder einzigartige Knoten in einem verteilten System ein potenzielles Bottleneck ist, da die Präsenz eines speziellen Knotens die Durchsatzrate senken und die Ausfallsicherheit des Gesamtsystems verringern kann. Diese Herausforderungen müssen natürlich mit einem klugen Algorithmus ausgeglichen werden, womit sich diese Arbeit auch befasst.

In der Zukunft sollen Systeme wie Drohnenschwärme selbstorganisierend und autonom agieren. Laut den Forschungsergebnissen der Technischen Hochschule Lausanne (EPFL) sind allerdings mit aktuellem Stand der Technik in der Robotik 100% selbstorganisierende Systeme weniger effizient als die Systeme, in denen Kommunikation zwischen Knoten möglich ist. Das heißt, dass die Synchronisation von Knoten im Schwarm eine gültige und in der Branche durchaus akzeptierte Lösung ist.

2.3 Analyse von Algorithmen für verteilte, ausfallsichere Systeme

Wie bereits im vorherigen Kapitel festgestellt, ist für das Funktionieren eines Schwarms die *Replikation* von Daten zwischen den Knoten erforderlich. Diese Aufgabe umfasst das Themengebiet der verteilten Systeme. Die Auswahl eines Algorithmus beeinflusst die Robustheit des Gesamtsystems und wird in diesem Kapitel betrachtet.

Verteilte Systeme finden in mehreren Teilen der IT-Infrastruktur Einsatz. Häufige Einsatzbereiche sind:

1. Dateisysteme
2. Datenbanken
3. Key-Value Stores (Schlüssel-Werte-Datenbank oder eine verteilte Hashtabelle)
4. Queues (Verarbeitung von Datenströmen, keine einfache Datenstruktur in diesem Kontext)

Diese genannten Systeme haben zwar unterschiedliche Anwendungsgebiete, die potenziell auftretenden Probleme sind allerdings sehr ähnlich:

Fehlertoleranz. Um die Fehlertoleranz zu garantieren, muss ein System mehrere *Replica* aller Daten besitzen und beim Ausfall der *Hauptreplica* auf eine andere reibungslos und ohne Datenverlust umsteigen können. Die Anzahl von Replicas soll immer eine ungerade Zahl sein, typischerweise werden drei oder fünf Replicas gebildet. Eine ungerade Zahl verhindert eine gleiche Aufteilung von Stimmen zwischen 2 Gruppen von Replicas. Gibt es eine Mehrheit aller Knoten, die den gleichen Wert haben, werden diese Daten als korrekt gewertet.

Zeit für Wiederherstellung. Ein Algorithmus für das verteilte System muss die Zeit für die Wiederherstellung verkürzen und intelligent auf mögliche Ausfälle reagieren. Der Algorithmus muss das Cluster unter neuen Einschränkungen umstrukturieren.

Erreichbarkeit. Die Erreichbarkeit eines Systems stellt fest, ob ein Knoten als abgestürzt oder als laufend betrachtet werden soll. In verteilten Systemen gibt es allerdings keine Möglichkeit zu überprüfen, ob ein Knoten abgestürzt ist oder das Netzwerk dieses Knotens überfordert ist. In modernen Systemen wird das Problem mittels *Timer* gelöst. Nach einem bestimmten Timeout wird der Knoten als nicht mehr erreichbar markiert. Nach welcher Zeit ein Knoten als nicht erreichbar betrachtet wird, ist für jede Konfiguration des Clusters spezifisch.

Skalierbarkeit und Durchsatzrate. Die Skalierbarkeit und Durchsatzrate in verteilten Systemen wird durch "Scaling up" Verfahren erreicht, wenn zur Lösung eines Problems oder für eine bessere Fehlertoleranz mehrere Knoten zusammengebracht werden. Für ein klassisches "Scaling up" Verfahren braucht man dagegen einen neuen, leistungsfähigeren Rechner, der alle Anfragen abarbeiten kann. Eine weitere Lösungsmöglichkeit stellen Algorithmen dar. Ein Algorithmus, der Leseoperationen von allen Knoten eines Clusters erlaubt, ist viel effizienter als der Algorithmus, der Leseoperationen nur von einem einzigen Knoten im Cluster erlaubt. Dieser einzige Knoten ist leicht zu überfordern und wird durch eine Hohe Nutzung des Netzwerks öfter als andere Knoten ausfallen.

Synchronisation und Timestamping. Timestamping hilft bei der Auflösung von Konflikten, beispielsweise zwischen Einträgen auf unterschiedlichen Knoten einer verteilten Datenbank bzw. eines verteilten Dateisystems. Der Eintrag mit dem höheren Timestamp hat Vorrang und soll einen anderen Eintrag überschreiben. Das Problem mit den Timestamps ist die Desynchronisation von Uhren auf allen Knoten. Mit modernen Technologien ist eine vollständige Synchronisation zwischen allen Knoten nicht umsetzbar. Selbst für Google, das mehrere Datenzentren mit Atomuhren und GPS-Fehlerkorrektur betreibt, ist das derzeit unmöglich. Die *TrueTime API* liefert zwar für höchst genaue Timestamps einen Zeitabstand, diese sind allerdings nicht eindeutig.

Eine weitere Lösung bietet die sogenannte *„Logische Anordnung von Operationen"*, welche Algorithmen zur Verfügung stellt. Es werden logische Zusammenhänge zwischen Operationen festgestellt und dadurch eine Linearisierung jener Operationen erreicht. (Anordnung dieser Operationen, als ob sie nacheinander ausgeführt wären).

Bis auf die Erreichbarkeit können all diese Probleme durch die Kombination spezialisierter Algorithmen gelöst werden. Die Replikation von Daten zwischen mehreren Knoten eines Schwarms fällt in die Klasse der Konsens-Algorithmen. Diese Klasse von Algorithmen löst ein fundamentales Problem der verteilten Systeme und Multiagentensysteme: die allgemeine Zuverlässigkeit des Systems beim Vorhandensein einer größeren Anzahl fehlerhafter Knoten. Dies erfordert eine Koordinierung der Knoten, um einen Konsens zu erreichen bzw. sich über einen Wert zu einigen, der für eine Rechenoperation nötig ist. Mögliche Anwendungen von Konsens-Algorithmen umfassen:

1. Anordnung von Transaktionen einer Datenbank
2. Cloud Computing
3. Lastverteilung (wie Load Balancing für Internetdienste)
4. Blockchain
5. Steuerung von UAVs (und generell mehreren Robotern / Agenten)

Um die richtige Entscheidung bei der Auswahl eines Algorithmus für den Konsens zu treffen, muss man zuerst die dahinter liegenden theoretischen Grundlagen verstehen, die auch Auswahlkriterien für Konsens-Algorithmen darstellen.

2.3.1 Konsens in verteilten Systemen

Der Konsens über einen Wert erfordert eine Einigung zwischen einer Reihe von Knoten. Einige der Knoten können ausfallen oder auf andere Weise unzuverlässig funktionieren. Deswegen muss der Konsens-Algorithmus fehlertolerant sein. Die Knoten haben die Aufgabe, die geltenden Werte zu ermitteln, miteinander zu kommunizieren und sich auf einen einzigen Wert zu einigen.

Konsens ist ein grundlegendes Problem bei der Steuerung von Multiagentensystemen. Definition von Konsens besagt, dass alle Knoten sich auf einen Mehrheitswert einigen müssen. In diesem Zusammenhang ist es erforderlich, dass sich im verteilten System eine Mehrheit bildet. Die Mehrheit ist immer um einen Knoten größer als die Hälfte aller Knoten, die am Wahlprozess teilnehmen. Ein oder mehrere fehlerhafte Knoten können das resultierende Ergebnis jedoch so verzerren, dass möglicherweise kein Konsens oder ein falsches Ergebnis erzielt wird.

Algorithmen, die den Konsens lösen, wurden entwickelt, um mit einer begrenzten Anzahl fehlerhafter Knoten umzugehen. Ein Knoten wird als fehlerhaft bezeichnet, wenn in diesem Knoten ein Fehler auftritt, durch den der Knoten nicht mehr erreichbar wird. Diese Algorithmen müssen eine Reihe von Anforderungen erfüllen, um nützlich zu sein. Ein Konsens-Algorithmus, der fehlerhafte Knoten toleriert, muss folgende Eigenschaften erfüllen:

Termination. Jeder korrekte Knoten entscheidet sich für einen Wert.

Integrity. Wenn alle korrekten Knoten denselben Wert vorschlagen, muss sich jeder korrekte Knoten für diesen Wert entscheiden.

Agreement. Jeder korrekte Knoten muss sich auf den gleichen Wert einigen.

All diese Kriterien werden durch die grundlegende Eigenschaften verteilter Systeme geprägt. Von diesen Eigenschaften zeichnet man vor allem aus:

Atomarität. Die wichtigste Eigenschaft aller verteilten Systeme besteht darin, dass ein System auf einen externen Beobachter wie ein einziger Rechner wirken soll. Das System verbirgt Implementierungsdetails und dient als eine Abstraktionsschicht.

Widerstandsfähigkeit. Wie es schon erwähnt wurde, braucht ein System mehrere Replicas, um einem Ausfall zu widerstehen. Die Widerstandsfähigkeit, also die Anzahl der Fehler, die ein System überleben kann, wird mit folgender Formel ausgerechnet:

$$T = (n - 1)/2$$

bei der

T : Anzahl der überstandenen Fehler
n : Anzahl der Knoten im Cluster

Nach einem sogenannten *„brain split“* , also wenn eine Verbindung zwischen zwei Rechenzentren (oder Clusters bzw. Gruppen von UAVs) unterbrochen wird, wird nur das Rechenzentrum, das über die Mehrheit aller Rechner verfügt, weiter aktuelle Daten liefern.

Neben den Eigenschaften müssen auch Bausteine für Modellierung verteilter Systeme in Betracht gezogen werden. Zu den grundlegendsten und für das Konsensproblem am relevantesten gehören:

State Machine Replication (Replikation eines Zustandsautomaten). In der Informatik ist die Replikation eines Zustandsautomaten eine allgemeine Methode zur Implementierung eines fehlertoleranten Dienstes durch Replikation von Knoten, Koordinierung der Clients und der Replicas. Der Ansatz definiert auch die Richtlinien für das Verständnis und die Entwicklung von Algorithmen für Replikation der Zustandsautomaten.

Die Funktionsweise eines replizierten Zustandsautomaten erfordert folgende Schritte:

1. Installation der Zustandsautomaten auf mehrere unabhängige Knoten
2. Empfang der Client Anfragen, die als Eingabedaten für den Zustandsautomaten dienen
3. Anordnung der Eingabedaten
4. Anwendung der Eingabedaten nach der ausgewählten Reihenfolge auf jedem Knoten
5. Den Clients mit den Ausgabedaten des Zustandsautomaten antworten
6. Beobachtung und Überprüfung der Replicas auf Zustandsabweichnungen oder Abweichungen in den Ausgabedaten

Die Replikation eines Zustandsautomaten ist ein rein theoretisches Konzept, es teilt aber das große Problem der fehlertoleranten Replikation in Subprobleme auf.

Bei der Lösung dieser Subprobleme können gewisse Kompromisse getroffen werden, welche sich dann als betriebliche Beschränkungen eines Produktivsystems bemerkbar machen könnten. Der 2. Schritt braucht beispielsweise einen Mechanismus, um die Nachrichten zuverlässig zu senden (*Atomic Broadcast*). 3. Schritt - ein theoretisches Framework zur Linearisierung aller Operationen (*eine lineare Anordnung aller Operationen*, als ob sie sequenziell und nicht parallel angewandt wurden, was für hochbelastete verteilte Systeme sehr problematisch ist), 4. Schritt - Definition *eines Zustandsautomaten*, 6. Schritt - ein Algorithmus für Fehlererkennung.

Atomic Broadcast (auch Total Order Broadcast). Es ist ein Broadcast, bei dem alle korrekten Knoten Nachrichten in derselben Reihenfolge empfangen. Die Sendung wird als „atomar“ bezeichnet, weil sie entweder an allen Knoten korrekt ankommt oder alle Teilnehmer das Senden abbrechen. Das beschreibt keine spezifische Implementierung, es ist lediglich ein Konzept und ein wichtiger Bestandteil des verteilten Rechnens. Die drei wichtigsten formalen Eigenschaften des Atomic Broadcast sind:

1. *Validity* (Gültigkeit). Wenn ein korrekt funktionierender Knoten eine Nachricht abschickt, dann empfängt ein anderer Knoten die Nachricht unvermeidlich.
2. *Agreement* (Einigung). Wenn ein korrekt funktionierender Knoten eine Nachricht empfängt, dann empfangen die Nachricht auch alle anderen Knoten.
3. *Total order* (Anordnung). Die Anordnung aller empfangenen Nachrichten ist bei allen Knoten gleich.

In der Realität kann Atomic Broadcast nicht implementiert werden, da die Netzwerke, in denen verteilte Systeme funktionieren, nicht zu 100% zuverlässig sind, selbst wenn der Netzbetreiber sich sehr bemüht, oder Pakete ein privates Netzwerk nicht verlassen. Betrachtet man Atomic Broadcast nüchtern, muss er in zwei Subkonzepte aufgeteilt werden: *Reliable Broadcast* und *Konsens*. Diese beiden Themen bilden den Schwerpunkt dieses Kapitels.

Reliable Broadcast (zuverlässiger Broadcast). Das Ziel dieses Broadcasts ist einfach: eine Nachricht zuverlässig über einen Cluster zu verbreiten, sodass alle Knoten die Nachricht empfangen und genau einmal verarbeiten. *„Zuverlässig“* bedeutet, dass es auch dann funktionieren sollte, wenn die Verbindungen zwischen Knoten fehlerhaft sind. Eine wichtige Annahme beim Reliable Broadcast lautet: wenn ein Prozess abstürzt und nicht mehr erreichbar ist oder die Netzverbindung einen Fehler erzeugt, wird irgendwann in der Zukunft dieser Prozess erneut gestartet bzw. das Netz wird wieder in Betrieb genommen und nach einem Versuch wird die Nachricht letztendlich zugestellt. Reliable Broadcast ist einfach zu implementiert und übernimmt nur zwei Eigenschaften des Atomic Broadcasts: *Validity* und *Agreement*. Total Order ist die Aufgabe des Konsenses.

Consensus (Konsens). Wie am Anfang dieses Kapitels schon erwähnt, erfordert Konsens die Einigung zwischen Prozessen über die Reihenfolge der in den Cluster geschriebenen Werte. Wären alle Knoten und das Netzwerk zu 100% korrekt und zuverlässig, bräuchte man den Konsens nicht.

Diese theoretischen Bausteine sind Grundlagen für zwei Algorithmen, die für die Auflösung des Konsenses in verteilten Systemen besonders wichtig sind: *Paxos* und *Raft*.

2.3.2 Paxos und Raft für Konsens in verteilten Systemen

Das Wichtigste bei Konsens-Algorithmen ist es zu verstehen, dass:

1. Konsens die Eignung auf einen einzelnen Wert bedeutet.
2. der Konsens erreicht ist, wenn sich die Mehrheit auf einen Wert einigt.
3. jeder Knoten im System den erreichten Konsens irgendwann erfahren muss.
4. die Kommunikationskanäle fehlerhaft sind, was die Erreichung des Konsenses verhindert.

Generell gibt es zwei Klassen von Systemen, die auf Konsens-Algorithmen beruhen:

Leader-Replica Schema. In diesem System gibt es immer einen *Leader*, der Anfragen beantwortet und Schreiboperationen steuert und Replicas (Followers), die Daten eines Leaders replizieren. In diesem Fall müssen alle Knoten nur dann einen Konsens erreichen, wenn der alte Leader abstürzt oder aus einem anderen Grund unerreichbar wird. In der Zwischenzeit, während noch kein neuer Leader ausgewählt wird, ist das System für neue Anfragen nicht erreichbar.

Peer-to-Peer Schema. In diesem System verhalten sich alle Knoten gleich und sind gleichrangig. Wenn mehrere Clients einige Werte schreiben wollen, müssen sich die Knoten untereinander einigen, in welcher Reihenfolge neue Werte im System gespeichert werden. Das ist wichtig, da alle Knoten dieselbe Reihenfolge gespeicherter Werte einhalten müssen. In diesem Fall muss der Konsens kontinuierlich aufrechterhalten werden.

Paxos und Raft gehören zur *„Leader-Replica Schema“* Klasse von Konsens-Algorithmen. Konsens-Algorithmen haben typischerweise folgende Eigenschaften:

1. Sie sorgen für Sicherheit unter allen *nicht byzantinischen Bedingungen* [1], einschließlich Netzwerk-Verzögerungen, Partitionen des Netzwerks, Paketverlust, Deduplizierung und Neuordnung der Pakete. Sicherheit in diesem Kontext bedeutet, dass das System immer einen korrekten Wert zurückgibt. Ein korrekter Wert in verteilten Systemen ist der Wert, der auf einem *Quorum* aller Knoten repliziert wurde. Ein Wert, der auf einem einzigen Knoten gespeichert wurde und dann an einen Client zurückgeschickt wird, ist nicht korrekt.
2. Sie sind funktionsfähig, solange die meisten Knoten betriebsbereit sind und miteinander und mit Clients kommunizieren können. Somit kann ein typischer Cluster von fünf Knoten den Ausfall von zwei beliebigen Knoten tolerieren. Es wird angenommen, dass Knoten durch einen Absturz unerreichbar werden. Sie können sich später von einem zwischengespeicherten Zustand wiederherstellen und wieder dem Cluster beitreten.

[1] Das ist die Situationen, wenn ein Knoten sich abhängig vom Beobachter entweder als ein korrekter oder als ein fehlerhafter Prozess verhält.

3. Ein Konsens-Algorithmus hängt nicht vom *Timing* ab, um die Konsistenz sicherzustellen: fehlerhafte Uhren und extreme Nachrichtenverzögerungen können schlimmstenfalls zu Verfügbarkeitsproblemen führen. Eine Minderheit langsamer Knoten muss sich nicht auf die Gesamtsystemleistung auswirken.

2.3.3 Der Paxos-Algorithmus

Paxos ist eine Familie von Algorithmen zur Lösung des Konsenses in einem Netzwerk unzuverlässiger oder fehlbarer Knoten. Der Paxos-Algorithmus wurde erstmals 1989 vorgestellt und nach einem fiktiven legislativen Konsenssystem auf der griechischen Insel Paxos benannt. Es wurde später als Artikel im Jahre 1998 veröffentlicht. Der ursprüngliche Algorithmus des Erfinders Leslie Lamport nennt man üblicherweise *Singledecree-Paxos*.

Die Paxos-Algorithmen umfassen ein Spektrum von Kompromissen zwischen der Anzahl der Knoten, der Anzahl der Round-Trip-Messages, der Aktivitäten einzelner Knoten, der Anzahl der gesendeten Nachrichten und den möglichen Fehlern. Obwohl kein deterministischer, fehlertoleranter Konsens-Algorithmus den Fortschritt in einem asynchronen Netzwerk garantieren kann, sichert Paxos Safety (das System gibt immer einen korrekten Wert zurück).

Beim Singledecree-Paxos hängen die Operationen, die Knoten ausführen können, von ihren Rollen ab:

Client. Ein Client schickt eine Anfrage an das verteilte System und wartet auf eine Antwort.

Acceptor (Akzeptor). Acceptors fungieren als fehlertoleranter *„Speicher“*. Acceptors werden in Gruppen gesammelt, die als Quorum bezeichnet werden. Jede an einen Acceptor gesendete Nachricht muss an ein Quorum von Acceptors gesendet werden. Jede Nachricht, die von einem Acceptor abgeschickt wird, hat keine Auswirkung, solange die gleichen Nachrichten nicht vom Quorum der Acceptors abgeschickt wird.

Proposer (Vorschlagender). Ein Proposer überzeugt die Acceptors, den von einem Client empfangenen Wert zu akzeptieren, sich darauf zu einigen und fungiert als Koordinator, um den Cluster bei Konflikten voranzutreiben.

Learner (Lernender). Ein Learner fungiert als der *Replikationsfaktor*. Sobald eine Client Anfrage von den Acceptors akzeptiert wurde, kann der Learner seine Rolle erfüllen (die Anfrage ausführen und eine Antwort an den Client senden). Um die Verfügbarkeit des Systems zu verbessern, können zusätzliche Learners hinzugefügt werden.

Leader. Paxos erfordert *einen speziellen Proposer*, um einen Fortschritt im Algorithmus und im ganzen Cluster zu erzielen. Paxos garantiert einen Fortschritt allerdings nur dann, wenn einer von den Proposers schließlich zum Leader ausgewählt wird. Wenn zwei Prozesse glauben, dass sie führend sind, können sie den Algorithmus blockieren, indem sie ständig widersprüchliche Aktualisierungen vorschlagen. Die Sicherheitseigenschaften bleiben in diesem Fall jedoch erhalten.

Das Konsensverfahren in Paxos hat folgende wichtige Meilensteine:

Prepare (Vorbereiten). Ein Proposer erstellt eine Nachricht, die "Prepare" genannt wird und die man mit einer Zahl N identifiziert. Dieser Wert muss unter allen anderen Proposers einzigartig sein. Beispielsweise kann der erste Proposer aus den Zahlen 1, 4, 7, ... auswählen, der zweite Proposer aus 2, 5, 8, ... , der dritte Proposer aus 3, 6, 9, ... Anschließend schickt der Proposer die Nachrichten an ein Quorum von Acceptors.

Promise (Versprechen). Jeder Acceptor wartet auf eine Nachricht von jedem Proposer. Wenn ein Acceptor eine Preparenachricht empfängt, muss der Acceptor die Identifikationsnummer N der gerade empfangenen Preparenachricht betrachten. Dabei gibt es zwei Fälle. Wenn N höher ist als jede vorher vorgeschlagene Zahl, muss der Acceptor Promise an den Proposer zurückschicken, andernfalls wird der Vorschlag ignoriert. Wenn der Acceptor eine Acceptnachricht zu einem bestimmten Zeitpunkt in der Vergangenheit angenommen hat, muss er die vorher vorgeschlagene Zahl und den entsprechenden akzeptierten Wert an den Proposer schicken.

Accept (Akzeptieren). Wenn ein Proposer ein Versprechen von einem Quorum aller Knoten erhält, sendet er eine Acceptnachricht mit der Zahl N (wurde in der Prepare Nachricht verwendet) und einem Wert, der vom Proposer ausgewählt wird. Wenn Acceptors bereits einen Wert akzeptiert haben, kann der Proposer nur einen zuvor ausgewählten Wert senden. Das beschreibt die wesentliche Einschränkung des ursprünglichen Singledecree-Paxos, die in Multidecree-Paxos (auch bekannt als multi-Paxos) gelöst wird.

Accepted (Akzeptieren). Wenn ein Acceptor eine Accept-Nachricht von einem Proposer erhält, muss er diese Nachricht nur dann akzeptieren, wenn er einem anderen Proposer mit einer höheren Prepare-Zahl noch kein Versprechen gegeben hat. Der Acceptor sollte den akzeptierten Wert registrieren und eine akzeptierte Nachricht an den Proposer und alle Learners senden.

Wenn Acceptors eine Accept-Anfrage eines Proposers erhalten, bestätigt das den vorgeschlagenen Wert und das Leadership des Proposers. Daher wird Paxos verwendet, um einen Wert und einen Leader in einem Cluster auszuwählen.

In Paxos kann ein einzelner Knoten eine Rolle, mehrere Rollen oder alle Rollen gleichzeitig übernehmen. Um einen Wert zu akzeptieren, muss jeder Knoten wissen, wie viele Knoten ein Quorum beinhaltet. Darüber hinaus können Paxos-Knoten in der ursprünglichen Version des Algorithmus nicht vergessen, welchen Wert sie früher ausgewählt haben. Es sollte allerdings erwähnt werden, dass es einige Variationen des Paxos-Algorithmus gibt, die das Löschen akzeptierter Werte erlauben.

Ein Durchlauf des Paxos-Algorithms erlaubt nur einen Wert zu wählen. Um dies zu tun, muss der Algorithmus nochmal gestartet werden. Diesen Algorithmus nennt man *Multi-Paxos* (Abkürzung von Multidecree-Paxos), der sich dem Raft-Algorithmus sehr ähnelt.

Bekannte Produktivsysteme und Werkzeuge, die Paxos als Konsens-Algorithmus verwenden, sind:

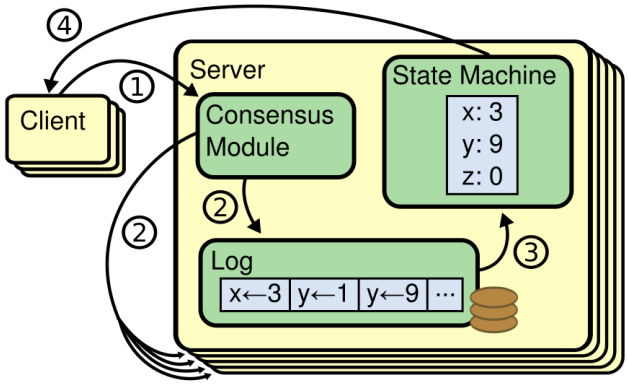


Abbildung 2.1: Architektur eines replizierten Zustandsautomaten

1. Amazon ECS
2. Apache Cassandra
3. Ceph
4. Neo4j

2.3.4 Der Raft-Algorithmus

Raft ist ein Konsens-Algorithmus zum Verwalten *eines replizierten Logs*. Er erzeugt ein Ergebnis, das dem (multi-)Paxos entspricht. Er ist ebenfalls genau so effizient wie Paxos, aber seine Struktur unterscheidet sich von ihm. Es ist wichtig zu beachten, dass Raft kein neuer Algorithmus ist, sondern eine Optimierung und eine inkrementelle Verbesserung eines bereits vorhandenen Algorithmus namens Paxos.

Konsens-Algorithmen entstehen typischerweise im Kontext replizierter Zustandsautomaten. Diese werden normalerweise mithilfe eines replizierten Logs implementiert, wie in Abbildung 2.1 gezeigt wird. Jeder Knoten speichert ein Log mit einer Reihe von Befehlen, die von seinem Zustandsautomaten bearbeitet wird. Jedes Log enthält die gleichen Befehle in der gleichen Reihenfolge, sodass jeder Zustandsautomat die gleiche Folge von Befehlen verarbeitet. Da die Zustandsautomaten deterministisch sind, berechnet jeder den gleichen Zustand und die gleiche Folge von Ausgaben.

Das replizierte Log konsistent zu halten ist die Aufgabe des Konsensus-Algorithmus. Das Konsensmodul an einem Knoten empfängt Befehle von Clients und fügt sie seinem Log hinzu. Es kommuniziert mit den Konsensmodulen an anderen Knoten, um sicherzustellen, dass jedes Log schließlich dieselben Werte in derselben Reihenfolge enthält, auch wenn einige Knoten fehlschlagen (siehe Abbildung 2.1).

Zu jedem Zeitpunkt ist jeder Knoten in einem von drei möglichen Zuständen sein:

Leader. Ein Leader verarbeitet alle Client-Anfragen. Wenn ein Client einen Follower kontaktiert, wird er von diesem Follower an den Leader weitergeleitet.

Follower (Anhänger). Followers sind passiv: sie selber schicken keine Anfragen aus und antworten nur auf mögliche Anfragen des Leader oder eines Kandidaten.

Candidate (Kandidat). Wenn ein Follower zum Kandidaten wird, darf er sich zum Leader vorschlagen, wobei er vom Quorum aller Knoten unterstützt werden muss.

Im Normalbetrieb es gibt genau einen Leader und alle anderen Knoten sind Followers.

Raft teilt die Zeit in Terms (Amtsperioden) beliebiger Länge. Terms werden mit aufeinanderfolgenden ganzen Zahlen nummeriert. Jeder Term beginnt mit einer Wahl, bei der ein oder mehrere Kandidaten versuchen, Leader zu werden. Wenn ein Kandidat die Wahl gewinnt, fungiert er für den Rest des Terms als Leader. Verschiedene Knoten können die Übergänge zwischen Terms zu verschiedenen Zeiten beobachten. In einigen Situationen übersieht ein Knoten möglicherweise eine Wahl oder sogar ganze Terms. Terms agieren in Raft als eine *„logische Uhr“* und ermöglichen es den Knoten, veraltete Informationen wie z.B. veraltete Leaders zu erkennen. Jeder Knoten speichert eine aktuelle Term-Nummer, die mit der Zeit monoton zunimmt. Der aktuelle Term wird ausgetauscht, wenn Knoten kommunizieren. Ein Term wird gleich nach dem Absturz des aktuellen Leaders beendet.

Um die Verständlichkeit zu verbessern, trennt Raft die Schlüsselelemente des Konsenses:

Wahlprozess des Leaders. Jeder neu gestartete Knoten wird standardmäßig zum Follower. Ein Knoten bleibt im Follower-Zustand, solange er einen *Heartbeat* von einem Leader erhält. Um eine Wahl zu beginnen, erhöht ein Follower den zuletzt gesehenen Term und wechselt in den Kandidatenzustand. Er stimmt dann für sich selbst und schickt RequestVote-Anfragen parallel an alle anderen Knoten im Cluster. Ein Kandidat bleibt in diesem Zustand, bis eine von drei Optionen vorkommt:

1. der Kandidat gewinnt die Wahl
2. ein anderer Knoten etabliert sich als Leader
3. ein Zeitraum vergeht ohne Gewinner
4. ein Kandidat gewinnt eine Wahl, wenn er Stimmen von einem Quorum für denselben Term erhält

Replikation des Logs. Sobald ein Leader gewählt wurde, beginnt er mit der Bearbeitung der Client-Anfragen. Jede Client-Anfrage enthält einen Befehl, der von den replizierten Zustandsautomaten ausgeführt werden soll. Der Leader hängt den Befehl als einen neuen Eintrag an sein Log an und schickt AppendEntries-Anfragen zu jedem anderen Knoten, um den Eintrag zu replizieren. Wenn der Eintrag sicher repliziert wurde (d.h. auf einem Quorum aller Knoten), wendet der Leader den Eintrag auf seinen Zustandsautomaten an und gibt das Ergebnis dieser Ausführung an den Client zurück. Jeder Logeintrag enthielt einen Befehl inklusive einer Term-Nummer.

Raft wurde als Basis für folgende Systeme genommen:

1. HashiCorp Consul
2. RethinkDB
3. Hazelcast
4. CockroachDB
5. etcd

2.3.5 Der Vergleich von Paxos und Raft

Um die Entscheidung zu treffen, mit welchem Algorithmus ein Produktivsystem implementiert werden soll, muss man sie anhand bestimmter Kriterien vergleichen. Die Algorithmen haben natürlich Ähnlichkeiten aber auch Unterschiede.

Ähnlichkeiten. Paxos und Raft sind schon deswegen sehr ähnlich, weil sie das Problem des Konsenses in verteilten Systemen lösen, was ein Subproblem des Atomic Broadcasts darstellt (Atomic Broadcasts wird in Probleme des Reliable Broadcasts und Konsenses zerlegt). Beide Algorithmen erfüllen Anforderungen an Safety, Availability und Zeitunabhängigkeit.

Sowohl Paxos als auch Raft versuchen, eine Gesamtreihenfolge von Befehlen zu bestimmen, die auf den replizierten Zustandsautomaten angewendet werden sollen. Diese Algorithmen lösen das gleiche Problem und verfolgen auch den gleichen Ansatz: beide wählen einen Leader, der Befehle in seinen Log hinzufügt und zu einem Quorum repliziert. Wenn ein Leader abstürzt, kommt ein Quorum zusammen und wählt einen neuen Leader. Beide Algorithmen garantieren, dass ein Knoten zum Leader ernannt wird, wenn er Stimmen von einem Quorum aller Knoten erhält. Außerdem haben Paxos und Raft immer nur einen Leader. Im Normalbetrieb hängt der Leader jede Operation an sein Log an und fordert andere Knoten auf, dasselbe zu tun. Sobald ein Quorum das getan hat, markiert der Leader den Logeintrag als *„commited“* (erfolgreich repliziert).

Wenn ein Follower glaubt, dass der Leader abgestürzt ist, wird der Follower zum Kandidaten und ersucht alle anderen Knoten für ihn zu stimmen. Wenn der Kandidat Stimmen von der Mehrheit aller Knoten (Quorum) bekommt, wird er zum Leader. Hier enden die Ähnlichkeiten beider Algorithmen.

Technische Unterschiede. Der größte Unterschied zwischen Paxos und Raft ist, wie die Safety Eigenschaft (always returning a correct result) von Paxos und Raft gewährleistet wird.

In Paxos wird ein Follower für jeden Kandidaten stimmen, der um eine Stimme ersucht. Daher muss die Antwort eines Followers alle bei dem Kandidaten fehlenden Logeinträge enthalten. Der Nachteil dieses Verfahrens ist die Belastung des Netzwerks, wenn ein Follower, der lange Zeit nicht erreichbar war, zu einem Kandidaten wird. Dieser Kandidat wird eine lange Liste von fehlenden Einträgen benötigen, was unter anderem eine langsame Zusammenführung aller fehlenden Einträge zur Folge hat. In Raft seinerseits wird ein Follower nur den Kandidaten anerkennen, der mindestens genauso einen aktuellen Log hat, wie der Follower. Generell wollte der Erfinder des Raft-Algorithmus die Einfachheit auf zwei Arten erreichen: die Anzahl von Nachrichten zu reduzieren und den Wahlprozess zu optimieren.

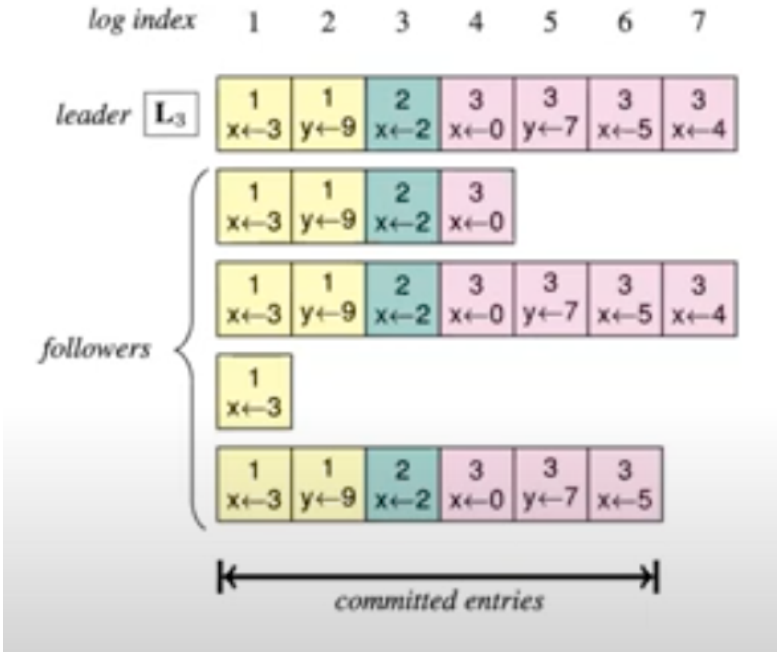


Abbildung 2.2: Zustand eines Logs.

In der Abbildung 2.2 wird ein Cluster mit fünf Knoten abgebildet. Der Knoten 1 ist der Leader, der Knoten 4 (mit einem einzelnen Eintrag im Log) ist ein Follower, der eine lange Zeit keine Updates von dem Leader erhalten hat. Laut dem Paxos-Algorithmus kann der vierte Follower auch zum Leader gewählt werden. Wenn er tatsächlich zum Leader gewählt wird, muss er alle fehlenden Logeinträge anordnen. Laut Raft also entweder der dritte oder fünfte Knoten. Nur diese beiden Knoten haben einen Log, der zumindestens genauso aktuell ist, wie der Log bei einem Quorum aller Knoten.

Nicht technische Unterschiede. Wenn man den Inhalt der ursprünglichen Publikation der Paxos- und Raft-Algorithmen vergleicht, wirkt der Raft-Algorithmus deutlich verständlicher und einfacher zu implementieren. Darauf hat der Autor des Raft-Algorithmus mehrmals hingedeutet. Das wurde auch durch eine Reihe von Publikationen bestätigt (Paxos Made Simple, Paxos Made Moderately Complex, Paxos Made Live), die versucht haben, den ursprünglichen Paxos-Algorithmus simpler zu erläutern. Es ist erwähnenswert, dass die erste von diesen Publikationen vom Autor des Paxos Algorithmus verfasst wurde.

Die Praxisorientiertheit und der Pragmatismus von Raft werden auch durch die Erwähnung bestimmter Technologien für den Nachrichtenaustausch hervorgehoben. Diese Technologie nennt sich *RPC* (Remote Procedure Call). In der Publikation des Paxos-Algorithmus gibt es keine solchen Tipps.

Der Vergleich in der Tabelle 2.1 zeigt deutlich, dass Raft durch seine Erklärbarkeit und pragmatische Darstellung einen frischen Blick auf moderne verteilte Systeme wirft und

	Raft	**Paxos**
Veröffentlichung	2014	1998
Erklärbarkeit des Algorithmus in der ursprünglichen Publikation laut dem Raft Artikel	Sehr gut	Mittelmäßig
Erklärbarkeit des Kerns des Algorithmus laut dem Raft Artikel	Gut	Gut
Spielraum bei der Implementierung	Weniger Spielraum. Einfach zu implementieren.	Mehr Spielraum. Viele Annahmen müssen getroffen werden.
Gewährleistung der Safety Eigenschaft	Nur der Kandidat mit dem aktuellsten Stand des Logs wird gewählt. Optimale Nutzung der Netzwerk Ressourcen.	Jeder Kandidat darf gewählt werden. Zusendung fehlender Logeinträge ist erforderlich. Nutzung der Netzwerkressourcen ist nicht optimal.

Tabelle 2.1: Vergleich der Raft und Paxos Algorithmen

die Optimierung der Replikation eine bessere Option für ein Produktivsystem darstellt. Das wird auch durch seine in den letzten Jahren schnell gestiegene Popularität bei großen kommerziellen Projekten (z.B. HashiCorp Consul) bestätigt.

2.4 Concurrency Modelle für die Umsetzung von verteilten Systemen

Algorithmen für verteilte Systeme umfassen mehrere Rechner, was ein passendes Concurrency Modell erfordert. Das Concurrency Modell bringt nicht nur die für die Entwicklung des Systems benötigten Bausteine und Abstraktionen, sondern auch ein *Kommunikationsmuster*. Dies wird in Form eines Frameworks und einer Bibliothek zur Verfügung gestellt bzw. direkt in der Programmiersprache implementiert. Dieses Kapitel beschäftigt sich mit diesen Aufgaben.

2.4.1 Concurrency in eingebetteten Systemen

Moderne Hardware- und Softwaresysteme verfügen über mehrere Funktionen. Eine Drohne muss sich z.B. im Raum lokalisieren, ein Modell vom Raum erstellen und die Aktuatoren steuern können. Zusätzlich muss sie mit einer Bodenstation oder mit der Cloud verbunden sein. Die von der Cloud oder von der Bodenstation empfangenen Daten und Signale könnten auch für die Steuerung von Aktuatoren notwendig sein. Das bedeutet, dass diese Funktionen miteinander zu kommunizieren können müssen.

Außerdem muss eine sich in einem Schwarm befindliche Drohne stets in der Lage sein, mit anderen Drohnen zu kommunizieren oder als Schwarm-Leader die von der Cloud

oder der Bodenstation empfangenen Daten an den Rest des Schwarms weiterzuleiten. Von Vorteil ist dabei, wenn das Programmiermodell für die Drohnen- und Serversoftware den gleichen Ansatz für die Verteilung von Rechenoperationen und bidirektionale Kommunikation verwendet.

Um diese Anforderungen erfüllen zu können, wird ein Concurrency Modell benötigt. Im Kontext der eingebetteten Systeme muss das Modell folgende Voraussetzungen erfüllen:

Fehlertoleranz. Ein Fehler in einer Komponente des Systems darf alle anderen Teile nicht zum Absturz bringen. Fehlerdomänen müssen klar definiert und abgekapselt werden. Die fehlerhafte Softwarekomponente muss ersetzt oder erneut gestartet werden.

Bidirektionale Kommunikation. Concurrent Components des Systems müssen miteinander kommunizieren, idealerweise ohne den intrinsischen Zustand zu teilen. *„Concurrent“* übersetzt auf Deutsch heißt *„nebenläufig“*, und bedeutet, dass zwei Teile eines Programms abwechselnd ausgeführt werden. Das darf nicht mit *„Parallelität“* verwechselt werden (*„parallelism“* auf Englisch), was das Ausführen von zwei Teilen eines Programms zur selben Zeit bedeutet.

Verteiltes Rechnen. Das Concurrency Modell soll im Idealfall den reibungslosen Aufruf von Remote-Services eines verteilten Systems ermöglichen. Das ist für die Verteilung von Tasks in einem Netzwerk von Drohnen und für eine freie Kommunikation mit der Cloud notwendig, dal ein Schwarm von Drohnen auch ein verteiltes IoT [2] System darstellt. Ein sogenanntes IoT System ist ein Netzwerk von physischen Objekten, die mit Sensoren, Aktuatoren, Kommunikationsmodulen und einer Recheneinheit ausgestattet sind. Eine Drohne verfügt über alle diese Komponenten.

2.4.2 Stand der Technik und Vergleich von Concurrency Modellen

Heutzutage existieren unterschiedliche Concurrency Modelle, die unterschiedliche Hintergründe und Motivationen für ihr Entstehen haben. Erste Modelle wurden 1970 vorgeschlagen und haben sich mit der Zeit geändert, wurden verbessert und an die Anforderungen moderner Software angepasst. Im Folgenden werden die Stärken und Schwächen der Modelle sorgfältig untersucht und verglichen.

Threads und Mutex. Dies ist ein klassisches Modell, welches in der Industrie schon seit Jahrzehnten im Einsatz ist. Dieses Modell ist auf den zwei Säulen der Concurrency aufgebaut: Thread und Mutex.

Thread. Ein Thread is ein Ausführungsstrang bzw. eine Ausführungsreihenfolge in der Abarbeitung eines Programms. Ein Thread ist Teil eines Prozesses. In Concurrenten Programmen (*„concurrent programs“* auf Englisch) können zwei Threads gleichzeitig auf eine geteilte Variable oder Ressource zugreifen. Wenn das passiert, ist es möglich, dass die Ressource später in einem

[2] Internet of Things oder Intrnet der Dinge

inkonsistenten Zustand bleibt. Um das zu vermeiden, müssen folgende Voraussetzungen erfüllt werden: der Zugriff auf alle geteilten Ressourcen muss synchronisiert sein. Sowohl der Schreibvorgang als auch der Lesevorgang muss Synchronisation verwenden. Das wird durch den wechselseitigen Ausschluss bzw. Mutex erreicht.

Mutex (Wechselseitiger Ausschluss). Der wechselseitige Ausschluss bezeichnet eine Gruppe von Verfahren, mit denen das Problem *des kritischen Abschnitts* gelöst wird. Das Problem tritt dann auf, wenn zwei Threads auf eine Ressource oder eine geteilte Variable zugreifen wollen. Das Mutex-Verfahren verhindert, dass Concurrent Prozesse bzw. Threads gleichzeitig oder zeitlich verschränkt gemeinsam genutzte Datenstrukturen unkoordiniert verändern, wodurch die Datenstrukturen letztendlich in einem inkonsistenten Zustand bleiben können. Die Faustregel für die Verwendung von Mutex lautet: die Kritische Sektion muss so klein wie möglich gehalten werden und so schnell wie möglich verlassen werden.

Da dieses Modell so einfach ist, bieten fast alle Programmiersprachen das Modell in unterschiedlicher Form an.

Die primäre Stärke des Threads- und Mutex-Modells ist eine breite Anwendbarkeit. Es kann für eine Vielzahl von Problemen eingesetzt werden. Alle abstrakten Modelle, wie z.B. das *Aktormodell* (actor model) und *CSP* (Communicating Sequential Processes), sind auf dessen Basis gebaut. Da Threads und Mutexes sehr nah an die Hardware gebunden sind, können sie sehr effizient sein, wenn sie richtig verwendet werden. Zusätzlich kann man sie sehr einfach in viele Programmiersprachen integrieren.

Außerhalb von einigen experimentellen Systemen mit verteiltem Arbeitsspeicher unterstützen Threads und Mutexes nur Architekturen mit lokalem RAM. Das heißt, Threads und Mutexes sind nicht geeignet für Rechenoperationen, die einen großen Arbeitsspeicher benötigen.

Aktormodell. Das Aktormodell ist ein universelles Concurrent Programmierungsmodell mit einer besonders breiten Anwendbarkeit. Es ist sowohl für lokale als auch für verteilte Speicherarchitekturen gedacht. Das Modell erleichtert geographische Verteilung und unterstützt Fehlertoleranz und Belastbarkeit. Ein Aktor ähnelt einem Objekt in einem objektorientierten Programm. Er sichert die Kapselung des Zustandes und kommuniziert mit den anderen Aktoren mittels Nachrichtenaustausch. Obwohl das Aktormodell ein generischer Ansatz für Concurrency ist, der fast mit jeder Programmiersprache verwendet werden kann, wird es sehr oft mit der Programmiersprache *Erlang* assoziiert.

Im Aktormodell ist ein Aktor die grundlegende Recheneinheit. Ein Aktor hat eine Reihe von Operationen, die er ausführen kann:

1. einen anderen Aktor erstellen
2. eine Nachricht schicken (die einzige Möglichkeit zum Kommunizieren im Aktormodell)
3. den inneren Zustand ändern (nur nachdem eine Nachricht empfangen wurde)

Die wichtigsten Bausteine eines Aktors sind folgende Bestandteile:

Mailbox. Eine der wichtigsten Eigenschaften des Aktormodells ist es, dass die Nachrichten asynchron ausgetauscht werden. Anstatt direkt an einen Aktor zugestellt zu werden, werden die Nachrichten in einer Mailbox gespeichert. Das bedeutet, dass die Aktoren von ihren Mailboxes entkoppelt sind und dass sie neue Nachrichten im eigenen Tempo bearbeiten. Dadurch werden sie nicht geblockt während des Nachrichtenaustauschs. Aktoren laufen zwar parallel zueinander, sie verarbeiten Nachrichten aber sequentiell in der [3] FIFO-Reihenfolge und gehen nur dann zur nächsten Nachricht über, wenn die Verarbeitung der aktuellen Nachricht abgeschlossen ist.

Adresse. Ein Aktor darf nur mit den Aktoren kommunizieren, deren Adressen er hat. Ein Akteur kann Adressen anderer Aktoren auf zwei verschiedene Arten erhalten: Adressen von Aktoren, die ein Aktor selber erstellt hat, oder Adressen von Aktoren, die Nachrichten an den Aktor geschickt haben. Ein Aktor kann sich mit mehreren Adressen identifizieren.

Aktor-Programme neigen dazu, defensive Programmierung zu vermeiden und die Philosophie *„let it crash“* einzuhalten, indem ein Supervisor eines Aktors stetig seine Exceptions (Ausnahmen) bearbeitet. Defensive Programmierung ist ein Ansatz, die sogenannte Fehlertoleranz zu erreichen. Hierbei wird versucht, mögliche Fehler zu antizipieren. Das hat mehrere Vorteile:

1. Der Code ist mit einer klaren Trennung zwischen dem *„happy path“* [4] und dem fehlertoleranten Code (Ausnahmebehandlungen) einfacher zu verstehen. Fehlertoleranter Code beschreibt den Vorgang, wenn ein untergeordneter Aktor abstürzt. Das umfasst die Delegierung oder den Neustart eines Aktors.
2. Aktoren sind voneinander getrennt und teilen keinen internen Zustand. Dadurch wird die Gefahr reduziert, dass Fehler in einem Aktor einen anderen Aktor beeinträchtigen. Insbesondere der *Supervisor* eines fehlerhaften Aktors kann nicht abstürzen. Die Funktion des Supervisors kann man mit der Funktion eines Managers vergleichen: er aggregiert mehrere untergeordnete Aktoren, sorgt dafür, dass sie ohne Absturz laufen, löst unberecherechenbare Probleme und - sofern notwendig - delegiert das Problem an einen übergeordneten Supervisor.

Einer der Hauptvorteile im Vergleich zu anderen Concurrency Modellen ist, dass das Aktormodell die Verteilung unterstützt. Das Senden einer Nachricht an einen Aktor auf einem anderen Rechner ist genauso einfach, wie eine Nachricht an einen lokalen Aktor zu schicken. Dabei ist zu beachten, dass die lokalen Aktoren voneinander isoliert sind und sie keinen Arbeitsspeicher miteinander teilen.

Das Aktormodell ist eines der am verbreitesten Programmierungsmodelle. Es bietet nicht nur Unterstützung für Concurrency, sondern auch Verteilung, Fehlererkennung und Fehlertoleranz. Als solches ist es sehr gut für die Art von Programmierproblemen geeignet, mit denen wir in der heutigen zunehmend verteilten Welt konfrontiert sind.

[3] First In First Out

[4] Unter „happy path“ versteht man im Kontext des Aktormodells die Geschäftslogik eines Programms oder sein Algorithmus

Funktionale Programmierung. Die Concurrent Programmierung in imperativen Sprachen - wie beispielsweise Java - ist aufgrund eines gemeinsamen veränderlichen Zustands schwierig. Funktionale Programmierung macht Concurrency einfacher und sicherer, indem gemeinsam genutzte veränderbare Zustände eliminiert werden. Auf unveränderliche Daten kann zugegriffen werden, ohne mehrere Threads zu stoppen. Das macht die Funktionale Programmierung so geeignet, wenn es um Concurrency und Parallelität geht. Funktionale Programme haben *keinen veränderlichen Zustand*, was durch folgende Konzepte erreicht wird:

Pure functions (Reine Funktion). Reine Funktion ist eine Funktion mit folgenden Eigenschaften: ihr Rückgabewert ist für dieselben Argumente immer gleich und ihre Auswertung hat keine Nebenwirkungen (Änderung einer globalen Variable). Somit ist eine reine Funktion ein rechnerisches Analogon einer mathematischen Funktion.

Referential transparency (Referentielle Transparenz). Ein Ausdruck wird als referentiell transparent bezeichnet, wenn er durch seinen entsprechenden Wert ersetzt werden kann, ohne das Verhalten des Programms zu ändern. Das bedeutet, dass eine Funktion keine Variablen aus einer globalen Umgebung verwenden darf. Ein Ausdruck, der nicht referentiell transparent ist, wird als referentiell *„opaque“* (nicht transparent) bezeichnet. Der Hauptnachteil von Sprachen, die referenzielle Transparenz erzwingen, besteht darin, dass sie die Operationen, die auf natürliche Weise zu einem imperativen Programmierstil gehören, umständlicher und weniger verständlich machen.

Reine Funktionen und Referentielle Transparenz ermöglichen Parallelisierung von Berechnungen oder den Betrieb in einer Concurrent Umgebung mit sehr wenig Aufwand. Da der Code im funktionalen Stil den veränderbaren Zustand eliminiert, sind die meisten Concurrency Fehler, die in traditionellen Thread und Mutex-basierten Programmen auftauchen, unmöglich. Es wird erwartet, dass der funktionale Code weniger effizient ist als das imperative Äquivalent. Obwohl es Auswirkungen auf die Leistung für einige Arten von Aufgaben hat, ist die Leistungsminderung üblicherweise relativ klein. Eine solch geringe Leistungsminderung wird meist zugunsten der verbesserten Skalierbarkeit und Robustheit in Kauf genommen.

Communicating Sequential Processes (CSP). CSP ist eine *Prozessalgebra* zur Beschreibung von Interaktionen zwischen kommunizierenden Prozessen. Die Idee wurde 1978 als imperative Sprache von Tony Hoare erstmals vorgestellt, dann von ihm zu einer formalen Algebra ausgebaut und 1985 mit der Veröffentlichung des Buchs mit dem gleichnamigen Titel *„Communicating Sequential Processes“* berühmt. Die Programmiersprachen Go [5] beinhaltet praktische Implementierungen der CSP und stellt - im Gegensatz zu vielen neuen prozeduralen und objektorientierten Programmiersprachen - kein Threading Modell für Concurrency zur Verfügung. Concurrency in Go ist ressourcenschonend und ist wesentlich einfacher zu verwalten als übliche Thread Pools. Als Basis dafür fungieren zwei Konzepte:

[5] Go ist eine kompilierbare Programmiersprache, die Nebenläufigkeit unterstützt und über eine automatische Speicherbereinigung verfügt. Entwickelt wurde Go von Google Mitarbeitern

Goroutines. Goroutine ist eine Funktion, die auf einem OS-Thread unabhängig von anderen Goroutinen läuft. Das bedeutet, dass ein OS-Thread mehrere Goroutinen abwechselnd ausführt. Goroutinen haben sehr kleinen Overhead für den Stack von nur ein paar Kilobytes.

Vorteile von Goroutinen werden klar, wenn man einen Thread Pool von Threads und einen Thread Pool von Goroutinen vergleicht. Thread Pool ist ein guter Weg, um CPU-gebundene Tasks auszuführen. Das sind die Tasks, die einen Thread für eine kurze Zeit aus dem Pool nehmen und am Schluss dem Thread zurückgeben. Wenn so ein Thread aus dem Pool für Netzwerkkommunikation ausgeliehen wird und der Netzwerkcall unbestimmt lang dauert, verschwindet der Nutzen des Thread Pools komplett. Wäre es ein Pool von Goroutinen, könnte man mit weniger Aufwand ressourcenschonender programmieren.

Channels. Ein Channel ist eine Pipeline, die zwei miteinander kommunizierende Goroutinen verbindet. Über einen Channel können Goroutinen strukturierte Daten austauschen. Ein Channel ist eine thread-safe Queue. Jede Goroutine mit einer Referenz auf die Queue kann Nachrichten an einer Seite einfügen und an der anderen Seite entfernen. Im Gegensatz zum Aktormodell, in dem Nachrichten von und an bestimmte Aktoren geschickt werden, muss hier der Sender den Empfänger nicht kennen und umgekehrt. Channel ist ein primärer Datentyp, der Go mit dem Stammbaum von CSP bezogenen Programmiersprachen verbindet.

Der Slogan von Go im Bezug auf Concurrency heißt “Do not communicate by sharing memory; instead, share memory by communicating.”

Laut Tabelle 2.2 erfüllt nur das Aktormodell die Anforderungen an das Concurrency Modell für ein Drohnensystem. Der Grund dafür ist, dass das Aktormodell standardmäßig Fehlertoleranz und verteiltes Rechnen unterstützt, ein direktes Teilen des Zustands verbietet und den Aufbau einer baumartigen Struktur von ausführbaren Einheiten ermöglicht. Jedoch müssen die Eigenschaften des Aktormodells noch in Detail betrachtet werden.

Das Actor Modell war kein Durchbruch in der Informatik, sondern eine inkrementelle Verbesserung von Ideen des *Lambda-Kalküls*, des Nachrichtenaustauschs und der Interprozesskommunikation. Das Lambda-Kalkül diente als eine Inspiration für den Interpreter der Programmiersprache Lisp, die ihrerseits den Austausch von geteilten Datenstrukturen ohne Parallelität ermöglicht hat. Danach war Simula 67 Pionier bei der Verwendung des Nachrichtenaustauschs für Berechnungen aller Art, motiviert durch diskrete Simulationsanwendungen. 1971 beschäftigte sich der Erfinder des Actor Modells mit der Programmiersprache Smalltalk-71 und war erstaunt von der Komplexität des Nachrichtenaustauschs. Das führte 1973 zur Veröffentlichung vom ersten Whitepaper zum Thema *„Actor Model“*. Das Actor Modell, das wir heute kennen, ist ein Ergebnis von einer Reihe von Dissertationen und wurde erst 1985 endgültig formalisiert.

Die grundlegende Herausforderung bei der Definition des Aktormodells ist, dass es keinen globalen Zustand bereitstellt. Das bedeutet, dass ein Rechenschritt innerhalb eines Aktors den Zustand von dem ganzen System nicht ändert. Der Zustand wird in dem

	Aktormodell	**CSP**	**Funktionale Programmierung**	**Threads und Mutex**
Primär Fokus	Fehlertoleranz und verteiltes Rechnen	Flexibilität und Ausdruckskraft	Nebeneffektefreiheit durch Immutability	Effizienz und breite Anwendbarkeit
Entkopplung von Producer-Consumer	-	+	nicht anwendbar	nicht anwendbar
Direktes Teilen des Zustands	-	-	-	+
Kommunikationsmuster	standardmäßig bidirektional zwischen den Aktoren	standardmäßig bidirectional zwischen den Coroutinen, unidirektional mit einer “receive-only” oder “send-only” Queue	nicht anwendbar	exklusive Nutzung von Arbeitsspeicherbereichen
Struktur von ausführbaren Codeeinheiten	Baum	flach	flach	flach
Hinzufügen neuer Nachrichten durch	Neue Match Cases innerhalb des Aktors	Weitergabe von neuen Queues in eine Goroutine	nicht anwendbar	nicht anwendbar
Fehlertoleranz by Design	+	-	-	-
Verteiltes Rechnen standardmäßig	+	-	-	-

Tabelle 2.2: Vergleich von Concurrency Modellen.

Fall zwischen mehreren in das System involvierten Aktoren geteilt. Da jeder Aktor seine eigenen lokalen Daten verwaltet, muss man sich um gleichzeitige Zugriffe auf diese Daten keine Sorgen machen. Deshalb wird das Aktormodell auch als *„shared-nothing“* Modell bezeichnet.

Die ursprüngliche Definition des Aktmodells befasste sich mit einem abstrakten Aktor ohne einer Anknüpfung zu einer bestimmten Technologie. In einem Aufsatz aus dem Jahr 2010 hat der Erfinder des Aktormodells Carl Hewitt das von ihm entwickelte Modell im Kontext der modernen verteilten Systemen (Rechenzentren und Webdienste) überdacht und noch folgende Herausforderungen gefunden:

Skalierbarkeit. Herausforderung, um Concurrency sowohl lokal als auch nicht lokal zu skalieren.

Transparenz. Überbrückung der Kluft zwischen lokaler und nicht lokaler Concurrency. Transparenz ist derzeit ein umstrittenes Thema. Einige Wissenschaftler befürworten eine strikte Trennung zwischen lokaler Concurrency mit Concurrent Programmiersprachen (z.B. Java und C#) und nicht lokaler Concurrency mit REST (Representational state transfer) für Webdienste. Eine klare Trennung schafft Probleme, wenn es wünschenswert oder notwendig ist, von lokalen Diensten auf Webdienste umzusteigen.

Inkonsistenz. Inkonsistenz ist ein häufiges Problem bei großen Informationssysteme. Zahlreiche Algorithmen aus dem Bereich verteilter Systemen sind dazu ausgerichtet, diese Inkonsistenzen zu eliminieren oder ihnen auszuweichen.

2.4.3 Umsetzung des Aktormodells in höheren Programmiersprachen

Im Allgemeinen gibt es für die Kapselung von Abstraktionen in der Informatik drei Ansätze: ein Framework, eine Bibliothek oder die direkte Umsetzung in der Programmiersprache. Das Aktormodell wird meistens in Form einer Bibliothek oder - in einigen Programmiersprachen - direkt implementiert.

Eine der Bekanntesten Implementierungen des Actor Models ist die Programmiersprache Erlang. Diese bringt 3 Konzepte zusammen: funktionale Programmierung, verteilte Programmierung und das Aktormodell. Joe Armstrong, der geistige Vater von Erlang, bezeichnet die Sprache am liebsten als *„nebenläufig ausgerichtete Programmiersprache“*, in der Prozesse die wichtigsten Objekte sind. Erlang ist eine der wenigen funktionalen Programmiersprachen, die in der Industrie eingesetzt wird. Insbesondere Telefon- und Netzwerkausrüster setzen Erlang aufgrund seiner guten Skalierbarkeit und Parallelität ein. Die Entwicklung von Erlang begann im Jahre 1986 bei Ericsson. 1998 enthielt der neue *Ericsson AXD 301 Switch* schon mehr als eine Million Zeilen von Erlang Code. Damals hat Erlang mit seinen Paradigmen *„alles ist ein Prozess“*, *„die einzige Kommunikation ist der Nachrichtenaustausch“*, *„keine lokale Fehlerbehandlung“* und *„reibungslose Updates ohne Ausfallzeit”“* für Ericsson die Verfügbarkeit von 99.999999999% (sogenannte “neun Neuens”) gewährleistet. 2014 berichtete Ericsson, dass Erlang in der Entwicklung von Software für GPRS, UMTS und LTE Netzwerke verwendet wurde. Die Telekommunikationsindustrie ist nicht das einzige Beispiel, wo Erlang erfolgreich eingesetzt wurde. Seit der Veröffentlichung als Open Source wird die Programmiersprache sehr erfolgreich bei Vocalink (ein Unternehmen von MasterCard), Klarna (Sofort Überweisungen) und WhatsApp eingesetzt.

Ein anderer Ansatz ist die Implementierung des Aktormodells in Form einer Bibliothek. Nach der auf Wikipedia veröffentlichten Liste von Actor Bibliotheken existieren mindestens 51 solcher Projekte, die in der aktiven Entwicklungsphase sind. Die fünf populärsten Bibliotheken sind in der Tabelle 2.3 aufgelistet.

Obwohl das Aktormodell schon 1970 definiert wurde, gewann es erst nach dem Entstehen von Erlang und Scala an Popularität (bzw. mit der Umsetzung des Aktormodells für diese Programmierungssprachen).

Was Scala im Bezug auf das Aktormodell auszeichnet, ist die Tatsache, dass es zwei von der Scala Community akzeptierte Implementierungen des Aktormodells gibt: die erste

Name	Programmiersprache	Sterne auf Github	Contributors auf Github
Ray	Python	13 900	1 200
Vert.x	Java	11 500	200
Akka	Scala	11 200	700
Orleans	C#	7 100	300
Actix	Rust	5 800	90

Tabelle 2.3: Popularität von Bibliotheken, die das Aktormodell implementieren (Stand 23.12.2020).

Implementierung ist wie ein Teil der Programmierungssprache umgesetzt, die zweite ist in Form einer Bibliothek realisiert und heißt *Akka*. Akka unterstützt unterschiedliche Modelle von Concurrency mit einem starken Fokus auf die Aktor-basierte Concurrency. Es ist nicht überraschend, dass die primäre Inspiration für Akka Erlang war.

Scala Aktoren. Laut der Grundidee von Aktoren in Scala, sollen sie nicht nur eine verständliche sondern auch eine leichtgewichtige Alternative zur üblichen parallelen Programmierung in der JVM mit Java Threads darstellen. Diese Leichtgewichtigkeit lässt sich jedoch nicht erreichen, wenn jeder Aktor durch einen Thread abgebildet wird, da Threads in der JVM einen sehr hohen Speicherverbrauch haben, um Threads anzuhalten, ihren Zustand zu speichern und fortzusetzen. Dieses sogenannte Context-Switching ist sehr aufwändig. Scala nutzt hierfür bspw. ein eigenes Threading-Modell und nicht das des darunterliegenden Betriebssystems. Eine Alternative zum thread-basierten ist ein ereignis-gesteuerter Ansatz, wofür man üblicherweise bestimmte Event-Handler in der Ausführungsumgebung registriert und diese beim Eintreten dieser Ereignisse aufgerufen werden. Dies nennt man *„Inversion of Control“*, da man den Kontrollfluss des Programms an die Umgebung abgibt. Diese entscheidet, wann diese Funktionen aufgerufen werden.

In Scala gibt es event-basierte Aktoren ohne inversion of control und Thread-basierte Aktoren. Beide Arten von Aktoren sind gemeinsam im Actor Trait definiert. Innerhalb der Empfangsmethode wird ein Pattern-Matching durchgeführt, um zu überprüfen, ob der Aktor diese Nachricht annehmen kann. Ist dies der Fall, wird die entsprechende Berechnung durchgeführt. Ist der Aktor mit der Verarbeitung einer Nachricht beschäftigt, kommt die gesendete Nachricht in eine Queue und wird später verarbeitet. Kann der Aktor die Nachricht nicht bearbeiten, wird sie verworfen.

Thread-basierte Aktoren eignen sich bspw. für I/O Operationen, welche ohnehin blockiert wären, wohingegen die event-basierten Aktoren in sehr großer Anzahl verwendet werden können. Um event-basierte Aktoren in so großer Anzahl zu ermöglichen, wird ein angehaltener Aktor, welcher bspw. in seiner receive Methode auf den Empfang einer Nachricht wartet, in Scala nicht durch einen Thread abgebildet. Weitere wichtige Eigenschaften von Akka und dem Aktormodell werden im nächsten Kapiteln behandelt.

2.4.4 Umsetzung des Aktormodells in eingebetteten Systemen

Unter den höheren Programmiersprachen existieren zahlreiche Frameworks für das Aktormodell und nebenläufige Berechnungen. Im Embedded Bereich, zu dem unbemannte Luftfahrzeuge gehören, schaut die Situation anders aus.

Abstraktionsschicht der Hardwaresprachen. Die Sprachen, die die Hardware von eingebetteten Systemen beschreiben und mit denen sie sich programmieren lassen, gehen mit sehr spezifischen Komponenten um, sind an Hardware sehr eng gebunden und stellen keine Abstraktionen zur Verfügung. Darüber hinaus werden diese Sprachen als sogenannte *Hardwarebeschreibungssprachen* bezeichnet. Das heißt, dass sie keine Programmiersprachen sind, obwohl sie Objekte beschreiben, deren Aufgabe meist die Informationsverarbeitung ist. Der Entwicklungsprozess für Embedded Geräte umfasst vor allem die Erstellung eines Modells für die Hardware und die Erzeugung einer textbasierten Beschreibung für das System. Ein leuchtendes Beispiel aus dieser Sprachfamilie ist die Hardwarebeschreibungssprache VHDL.

Hardwarebeschreibungssprachen werden an dieser Stelle erwähnt, weil die Probleme, die man mit Modellen für Concurrency in höheren Programmiersprache löst, in der Embedded Welt und bei der Beschreibung von Hardware mit sehr ähnlichen Abstraktionen in modellbasierten Design- und Simulationswerkzeugen gelöst werden. Das heißt, Modelle aus den beiden Welten (Softwareentwicklung und Systementwicklung) haben dieselben Wurzeln, aber die Interpretationen unterscheiden sich bei ausführlicher Betrachtung. Eines der Projekte, das zum Ziel hat, Modellierung, Simulation und das Design von nebenläufigen, eingebetteten und Echtzeitsystemen einfacher zu machen, ist *Ptolemy II*.

Modellierung. Ptolemy II ist ein Open Source Framework, das Experimente mit aktororientiertem Design unterstützt. Im Kontext von Ptolemy II sind Aktoren Softwarekomponenten, die gleichzeitig ausgeführt werden und über Nachrichten kommunizieren, die über miteinander verbundene Ports gesendet werden. Ein Modell ist eine hierarchische Verbindung von Aktoren. Ein Schwerpunkt des Projekts liegt darin, heterogene Kombinationen von Berechnungsmodellen, kontinuierlichen Zeitmodellen mit Zustandsmaschinen und synchronen/reaktiven Systemen mit Zustandsmaschinen zu verstehen.

Ptolemy II konzentriert sich auf die aktororientierte Modellierung komplexer Systeme und bietet einen Ansatz für nebenläufige Berechnungen. Der zentrale Begriff bei der hierarchischen Modellzerlegung ist die Domain, die ein bestimmtes Berechnungsmodell implementiert. Technisch dient eine Domain dazu, den Steuerungs- und Datenfluss zwischen Komponenten von der tatsächlichen Funktionalität einzelner Komponenten zu trennen.

Ptolemäus II ermöglicht die Verwendung einiger Domains, die im Systemdesign, der Modellierung und der Simulation verwendet werden:

Dataflow. Die Ausführung eines Aktors in der Dataflow-Domain besteht aus einer Folge von *„Firings“*, wobei jeder Firing als Reaktion auf die Verfügbarkeit von Eingabedaten erfolgt. Ein *Firings* ist eine für gewöhnlich kleine Berechnung, die die Eingabedaten verwendet und Ausgabedaten erzeugt. Dataflow

Modelle sind ideal für die Darstellung von Streaming Systemen, bei denen Sequenzen von Datenwerten in relativ regelmäßigen Mustern zwischen Komponenten fließen. Signalverarbeitungssysteme, wie Audio- und Videosysteme, passen besonders gut zusammen.

Prozessnetzwerke. In der Domain des Prozessnetzwerks stellen Aktoren Prozesse dar, die über FIFO-Queues (konzeptionell unendlicher Kapazität) kommunizieren. Das Schreiben in die Queues ist eine nicht-blockierende Operation, während das Lesen aus einer leeren Queue einen Reader Prozess blockiert. Die einfache Strategie des blockierenden Lesens und des nicht-blockierenden Schreibens gewährleistet den Determinismus des Modells. Ein Prozessnetzwerk eignet sich zur Beschreibung nebenläufiger Prozesse, die durch einen asynchronen Nachrichtenaustausch miteinander kommunizieren. Prozessnetzwerke kann man als das Aktormodell betrachten, das in höheren Programmiersprachen verwendet wird. Mit Prozessnetzwerken kann man auch die Vorgänge in verteilten Systemen darstellen. Der einzige Unterschied zur ursprünglichen Definition eines verteilten Systems besteht darin, dass die Nachrichten in der Versandreihenfolge zwar zugestellt werden, die genaue Zeit allerdings nicht vorher festgelegt ist. In der klassischen Modellierung von verteilten Systemen wird angenommen, dass alle Nachrichten eventuell zugestellt werden müssen, jedoch die Reihenfolge, in der sie ankommen, ist nicht definiert.

Rendezvous. Die Rendezvous-Domain ähnelt einem Prozessnetzwerk insofern, dass Aktoren gleichzeitige Prozesse darstellen. Im Gegensatz zur Semantik *„Fire And Forget“* von Prozessnetzwerken kommunizieren Aktoren in der Rendezvous-Domain jedoch durch den atomaren sofortigen Datenaustausch. Wenn ein Aktor Daten an einen anderen Aktor sendet, wird der Sender blockiert, bis der Empfänger empfangsbereit ist. Wenn ein Aktor versucht, Eingabedaten zu lesen, wird er auch blockiert, bis der Sender der Daten bereit ist, die Daten zu senden. Infolgedessen bleibt der Prozess, der als erster einen Treffpunkt erreicht, blockiert, bis der andere Prozess denselben Treffpunkt erreicht. In dieser Domain ist es auch möglich, Mehrweg-Rendezvous zu erstellen, bei dem mehrere Prozesse den Rendezvous Punkt erreichen müssen, bevor ein Prozess fortgesetzt werden kann. Die Rendezvous-Domäne ist besonders nützlich für die Modellierung von Problemen, bei denen eine einzelne Ressource von mehreren asynchronen Prozessen gemeinsam genutzt wird.

Die Tatsache, dass die Rendezvous-Domain auf der Prozessalgebra von Tony Hoare basiert, macht klar, dass das Communicating Sequential Processes Concurrency Modell und Rendezvous dieselben Wurzeln in der akademischen Forschung haben.

Discrete Event. In der Discrete Event Domain kommunizieren Aktoren über Events, die auf einer Zeitleiste platziert sind. Jedes Event hat einen Wert und einen Zeitstempel. Aktoren verarbeiten Events in chronologischer Reihenfolge. Die von einem Aktor erzeugten Ausgabeevents müssen nicht früher als die verbrauchten Eingabeevents vorliegen. Anders gesagt sind Aktoren in der Discrete Event Domain kausal.

Die Ausführung dieses Modells verwendet eine globale Event-Queue. Wenn ein Akteur ein Event generiert, wird das Event entsprechend seines Zeitstempels in die Queue eingefügt. Während jeder Iteration des Modells werden die Events mit dem kleinsten Zeitstempel aus der globalen Event Queue entfernt und der entsprechende Aktor wird ausgelöst.

Discrete Event eignet sich gut für die Modellierung des Verhaltens komplexer Systeme im Zeitverlauf. Es kann beispielsweise Netzwerke, digitale Hardware und Finanzsysteme modellieren.

Die oben aufgelisteten Modelle sind Beispiele davon, wie Hardwarebeschreibungssprachen und höhere Programmiersprachen verflochten sind. Die Anwendung von ähnlichen Abstraktionen bei der Systementwicklung ist sinnvoll und etabliert eine *„gemeinsame Sprache“* für Entwickler von Hardware- und Softwarekomponenten.

Am Beispiel des Process Network Modells für eingebettete Systeme sieht man, dass es dieselbe Wurzeln wie die Aktormodelle für höhere Programmiersprachen hat.

Viele von den heutzutage entwickelten Systemen kombinieren heterogene und oft komplexe Subsysteme. Ein modernes Auto kann einen komplexen Motor, elektronische Steuergeräte, Traktionskontrollsysteme, Karosserieelektronik (zur Steuerung von Fenstern und Türschlössern), Infotainmentsysteme, Klimatisierung und Belüftung sowie eine Vielzahl von Sicherheits-Subsystemen (wie Airbags) kombinieren. Jedes Subsystem wird mit einer Kombination aus Software, Elektronik und mechanischen Teilen realisiert. Die Entwicklung derart komplexer Systeme ist sehr anspruchsvoll, da zum Teil selbst kleinste Teilsysteme mehrere Ingenieurdisziplinen umfassen.

2.5 Fazit

Der Drohnenschwarm ist ein komplexes System, das aus mehreren Rechenknoten besteht, die Kommunikation, Synchronisation und Koordination für die Zusammenarbeit brauchen. Die Zusammenarbeit im Offline-Modus (ohne Verbindung zur Cloud) oder mit der begrenzten Erreichbarkeit des zentralen Servers ist nur dann möglich, wenn es im Schwarm einen Leader gibt, der die Konsistenz von Daten garantiert und sich mit der Cloud synchronisiert. Für die Lösung dieser Aufgabe gibt es mehrere Algorithmen, von denen sich besonders Raft mit seiner Erklärbarkeit und Einfachheit auszeichnet.

Für die Umsetzung eines verteilten Systems ist auch ein Concurrency Modell nötig. Verteilte Systeme stellen folgende Anforderungen an das Modell: standardmäßige Fehlertoleranz und Unterstützung des verteilten Rechnens. Von den Modellen, die in diesem Kapitel behandelt werden, ist das Aktormodell dafür am besten geeignet.

Eine gute Perspektive auf die Theorie und Modellierung von nebenläufigen, eingebetteten Systemen gibt Ptolemy II. Ptolemy II hat als ein Design-Werkzeug und als Framework ein sehr interessantes Problem aufgeworfen, was für die Entwicklung von aktorbasierten Systemen wichtig ist. Wie müssen die Aktoren aufgegliedert sein? Wie viele Aktoren muss es in einem System geben? Die Antwort lautet: wenn die Akteure relativ klein sind - also für jede Kommunikation weniger Berechnungen durchführen - kann das

Overhead von Multithreading und Kommunikation zwischen Threads die Leistungsvorteile der parallelen Ausführung überwältigen. Daher sollten Ingenieure Leistungsvorteile nur für große oder mittelgroße Aktore erwarten.

Kapitel 3

Entwicklung

Die im zweiten Kapitel beschriebenen theoretischen Grundlagen und auch praktischen Überlegungen zur Implementierung verteilter Systeme für Drohnenschwärme müssen jetzt in der Form eines Prototypen umgesetzt werden. Dies beginnt mit der Beschreibung der getroffenen Annahmen, da die Implementierung eines voll funktionsfähigen verteilten Systems in begrenzten Zeitrahmen mehr Zeitaufwand, Testing und Aufmerksamkeit für Details erfordert hätte. Dann folgt die Beschreibung des Gesamtsystems mit einer Gliederung der wichtigsten Komponenten. Anschließend wird die Implementierung des Raft-Algorithmus detailliert untersucht.

3.1 Annahmen und Einschränkungen in der Entwicklung

Die Implementierung jedes Systems erfordert Zeit, die auch exponentiell mit der Komplexität des Vorhabens steigt. Um einen realisierbaren Prototypen innerhalb einer überschaubaren Zeit zu implementieren, wurde eine Reihe von Annahmen getroffen.

3.1.1 Scala als Programmiersprache

Als Programmiersprache für die Implementierung des Prototypen wurde Scala verwendet. Scala ist eine *Universalsprache*, die sowohl objektorientierte Programmierung als auch funktionale Programmierung unterstützt. Die Sprache ist statisch typisiert. Der Scala-Quellcode wird in Java Bytecode kompiliert, sodass der resultierende ausführbare Code auf der Java Virtual Machine (JVM) ausgeführt werden kann. Scala bietet Interoperabilität mit Java. Im Gegensatz zu Java verfügt Scala über viele Eigenschaften funktionaler Programmiersprachen, wie beispielsweise Scheme, Standard ML und Haskell, einschließlich Currying, Immutability (Unveränderlichkeit), die Lazy Evaluation (bequeme oder auch faule Auswertung) und Pattern Matching. Es verfügt ebenfalls über ein fortschrittliches Typsystem, das algebraische Datentypen, Kovarianz und Kontravarianz sowie anonyme Typen unterstützt. Andere Funktionen von Scala, die in Java nicht vorhanden sind, sind das Überladen von Operatoren, optionale Parameter und benannte Parameter.

Laut *„Github Language Stats“* (Stand 23.12.2020) erreichte Scala bei der Anzahl an

Pull-Requests den elften Platz und überholte somit Programmiersprachen wie Rust, Kotlin, Swift und Haskell.

Für die Implementierung eines Produktivsystems, welches auf einer Drohne läuft und auch Leistungsanforderungen des Betriebssystems einer Drohne erfüllt, wären solche Systemprogrammiersprachen wie C++ oder Rust geeignet. Scala ermöglicht aufgrund seiner Simplizität ein schnelles Prototyping, das Aktorsystem (actors system) dieser Programmierungssprache mit zahlreichen Eigenschaften hat einen guten Ruf bekommen und die Integration des Aktorsystems mit dem Play-Web-Frameworks machen Scala zu einer guten Option für die Implementierung eines Prototypen.

3.1.2 Akka für das Implementieren verteilter Systeme

Den größten Vorteil von Scala bietet das Aktorsystem, genauer gesagt das Aktorsystem der Akka-Bibliothek. Die Verwendung von Akka macht die Erstellung der Infrastruktur für ein Aktorsystem und das Schreiben des Low-Level-Codes, der zur Steuerung des grundlegenden Verhaltens erforderlich ist, überflüssig.

Ein Aktor in Akka hat immer einen Elternknoten und alle Knoten sind Kinder des Guardian-Knotens, der beim Start eines Aktorsystems erstellt wird. Eine Instanz des ActorSystems dient zur Kommunikation mit dem Aktorsystem, zur Erstellung neuer Aktoren, zur Adressierung und zur Verwaltung des Kontexts. Eine reibungslose Kommunikation zwischen den Aktoren ist nur innerhalb des Systems möglich. Wenn eine andere Softwarekomponente einen Aktor innerhalb des Aktorsystems aufrufen möchte, muss diese Anfrage mittels Adapter übersetzt werden. Dasselbe gilt für Antworten des Aktorsystems. Die Kommunikation im Aktorsystem ist immer asynchron.

Bevor der erste Aktor gestartet wird, hat Akka bereits 2 Aktoren erstellt. Die Namen dieser eingebauten Aktoren enthalten *„guardian“* (Wächter). Die Guardian-Aktoren umfassen:

1. „/“ - der sogenannte Root-Guardian. Das ist das übergeordnete Element aller Aktoren im System und das letzte, das angehalten wird, wenn das System stoppt.
2. „/system“ - der System-Guardian. Das ist ein Aktor, der das System aufrechterhält. Akka oder andere Bibliotheken, die auf Akka basieren, können Aktoren im Systemnamespace erstellen.
3. „/user “ - der Benutzer-Guardian. Das ist der Aktor, der alle anderen Aktoren einer Anwendung startet.

Struktur der Guardian Aktoren wird in der Abbildung 3.1 dargestellt.

Anwendung des Akka Aktorsystems und seiner Module bringt folgende Vorteile für verteilte Systeme:

Protokollunabhängigkeit. Standardmäßig bietet Akka eine Integration mit HTTP, aber mittels Konnektoren (Adapters anders gesagt) kann man auch problemlos über MQTT, AMQP, JMS (Java Message Service) und gRPC kommunizieren.

Verteilte Aktoren. Akka Remoting ermöglicht Aktoren, die auf verschiedenen Rechnern laufen, Nachrichten nahtlos auszutauschen. Dank des Aktormodells sieht ein

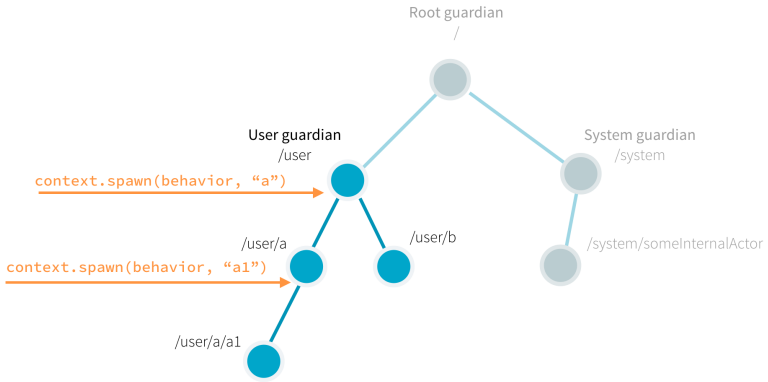


Abbildung 3.1: Beispiel eines Aktorsystems zusammen mit den Aktoren, die von Akka standardmäßig erstellt werden.

ferner und lokaler Nachrichtenversand gleich aus. Es bietet die Basis, auf der das Cluster-Subsystem aufgebaut ist.

Unterstützung von Clusters. Wenn es in einem verteilten System eine Reihe von Aktoren gibt, die zusammenarbeiten, um eine Aufgabe zu lösen, muss das System auf eine bestimmte Weise aufgebaut und gesteuert werden. Während Akka-Remoting das Problem der Adressierung und Kommunikation zwischen Komponenten eines verteilten Systems löst, bietet Clustering die Möglichkeit, diese Knoten in der Form eines Metasystems mit einem Mitgliedschaftsprotokoll zu organisieren. Das ist ein Teil der Kontrollebene eines Clusters.

Sharding (horizontale Fragmentierung). Sharding ist ein Vorgehen, das verwendet wird, um eine große Anzahl von Aktoren auf Knoten eines Clusters aufzuteilen und sie auch auf andere Knoten zu verschieben, wenn andere Knoten abstürzen oder unerreichbar werden.

Akka verfügt über mehrere Kommunikationspatterns. Einige von ihnen sind für Aktorsysteme spezifisch und dienen dazu, ein erklärbares und skalierbares System zu entwickeln. Diese spezifische Patterns sind z.B:

1. Zusendung des Ergebnisses eines Futures [1] an sich selbst
2. Per-Session-Child-Actor (Erstellung eines Aktors je Sitzung) in dem ein Elternknoten einen Kindknoten erstellt. Der Kindknoten stellt eine asynchrone Operation dar, die allen anderen Aktoren abfragen muss, das Ergebnis dem Elternknoten zur Verfügung stellt und sich anschließen stoppt. Die Erstellung eines neuen Aktors für jede Anfrage kann verschwenderisch klingen (vor allem in einem eingebetteten System), die Erstellung eines neuen Aktors erfordert allerdings keine Ressourcen des Betriebssystems und ist deswegen sehr billig.

Patterns, die bei der Implementierung des Raft-Algorithmus eingesetzt wurden, sind:

[1] Ein Future bezeichnet einen Platzhalter (Proxy) für ein Ergebnis, das noch nicht bekannt ist, meist weil seine Berechnung noch nicht abgeschlossen ist.

Fire and Forget (Tell-Ansatz). Fire-And-Forget ist der fundamentale Weg für die Kommunikation zwischen Aktoren. „Tell“ ist asynchron, was bedeutet, dass die Methode in einem anderen Thread gestartet wird. Nachdem der Befehl ausgeführt wurde, gibt es keine Garantie, dass die Nachricht noch vom Empfänger verarbeitet wurde. Das bedeutet auch, dass es keine Möglichkeit gibt herauszufinden, ob die Nachricht empfangen wurde, die Verarbeitung erfolgreich war oder fehlgeschlagen ist. Fire-And-Forget ist so üblich bei Akka, dass es einen speziellen symbolischen Namen hat: “!” (Ausrufezeichen). Der Versand einer Nachricht hätte dann so ausgesehen “actor ! message”.

Request-Response mit einer oder mehreren Antworten. Mehrere Interaktionen erfordern eine oder mehrere Antworten. Um die Antwort zu erhalten muss der Absender seine Adresse in der Nachricht mitschicken.

Request-Response mit einer einzelnen Antwort (Ask-Ansatz). Wenn für eine Anfrage genau eine Antwort erwartet wird, verwendet man Ask-Anfragen. Eine Ask-Anfrage hat ebenfalls eine spezielle Bezeichnung: “?”. Da alle Kommunikationen im Aktorsystem asynchron sind, darf man einen Aktor mit der Ask-Anfrage nicht blockieren. Die Antwort wird asynchron empfangen und in einen Callback übergeben, der bei der Absendung der Ask-Anfrage erstellt sein soll.

Interaktion mit externen Komponenten. Da Aktoren standardmäßig nur mit anderen Aktoren innerhalb eines Aktorsystems kommunizieren können, braucht man eine Brücke für die Außenwelt. Eine der möglichen Lösungen ist Future. Dies ist eine Abstraktionsschicht über übliche Callbacks und stellt Ergebnis einer asynchronen Operation dar. Ein Future wird normalerweise von einem Promise erstellt, der ein Proxy zwischen einer Rechenoperation und dem Future (dem asynchronen Ergebnis) darstellt.

Scheduling messages to self (Zusendung an sich selbst zu planen). Standardmäßig erlaubt Akka einem Aktor Nachrichten an sich selbst zu schicken. Dadurch kann man Timeouts implementieren, auf denen Paxos und andere Algorithmen für verteilte Systeme beruhen.

Das Kommunikationsmuster eines Knotens mit anderen Knoten und dessen Verhalten kann sich mit der Zeit ändern. Das passiert, wenn ein Knoten mehr Funktionen bzw. andere Funktionen in der Hierarchie übernimmt, beispielsweise die Funktion eines Beobachters und Workers gleichzeitig, oder die Funktion eines Beobachters und eines Response Aggregators. Da ein Aktor als solcher einen Zustandsautomaten darstellt, der beliebig kompliziert sein kann, führt es zu einem Zustandsautomaten mit verschachtelten Zuständen. Um diese Überlappung von Funktionen und Komplexitäten aufzulösen, gibt es bei Akka zwei Herangehensweisen: die erste klassische Methode, die sich vorzüglich bewährt hat, ist die Zersetzung eines Aktors in mehrere Aktoren. Die zweite Methode ist die Auswechslung von *Behaviors* (Verhalen). Ein Behavior repräsentiert einen Übergang in einen neuen globalen Zustand. Ein solcher globaler Übergang wäre in Raft der Übergang vom Follower-Zustand zum Leader-Zustand. Ein Aktor dient in diesem Fall als Container (Behälter) für Behaviors mit einem anfänglichem Verhalten. Behaviors entscheiden selbstständig, was das nächste Behavior ist und ob ein Behavior zu einem bestimmten Zeitpunkt überhaupt ausgetauscht werden soll. Wenn ein Behavior ausgewechselt wird, bleibt der Container (also Aktor) intakt. Einen Übergang

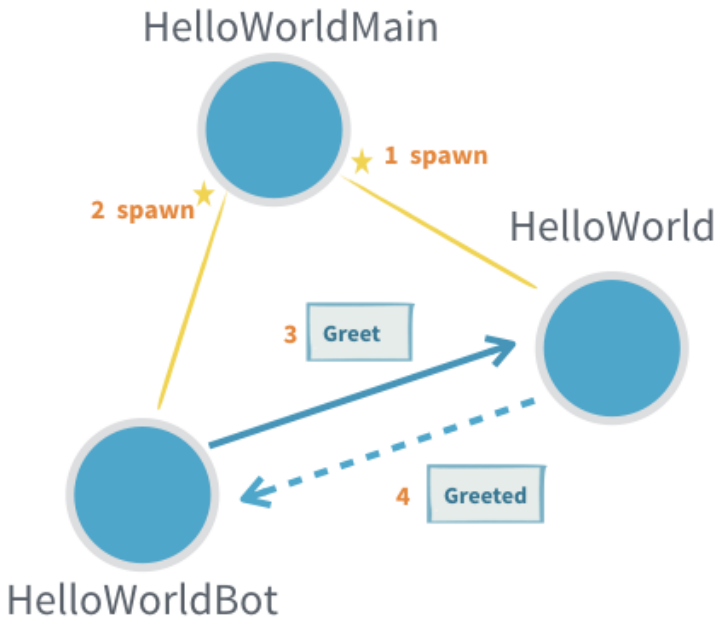


Abbildung 3.2: Übergang von Zuständen im Raft-Algorithmus

von Zuständen innerhalb eines Aktors könnte man beim Raft-Algorithmus wie in der Abbildung 3.2 darstellen.

Dadurch, dass Akka viele wichtige Bausteine für den Aufbau verteilter Systeme auf Mikroebene (automatische Serialisierung und Deserialisierung von Nachrichten, Timers, Austausch von Behaviors) und Makroebene (Protokollunabhängigkeit, Unterstützung verteilter Aktoren und Clusters von Aktoren) bietet, ist es für die Implementierung eines verteilten Systems besonders gut geeignet.

3.1.3 Einschränkungen bei der Implementierung eines verteilten Systems mit Akka

Wie es schon im zweiten Kapitel festgestellt wurde, folgt das Aktorsystem dem Prinzip *„let it crash“*, bei dem ein Kindknoten seinem Elternknoten über Ausnahmesituationen Bescheid gibt, sodass der letzte Knoten alle dieser Situationen auflöst. Wenn ein Elternknoten eine Ausnahmesituation nicht auflösen kann, delegiert er seinem Elternknoten die Verantwortung für die Situation. Dadurch entsteht ein Baum von Aktoren, in dem ein übergeordneter Knoten einen untergeordnete Knoten beobachtet und verwaltet. Das bedeutet, dass alle fehlerhaften Operationen in Blättern des *Observation-Tree* (Beobachtungsbaum) konzentriert werden. Je näher ein Aktor der Wurzel des Baumes ist, desto weniger Ausnahmesituationen werden für ihn möglich sein. Die Robustheit der Hierarchie wird durch die Abgrenzung von fehlerhaften Operationen in mehreren kleinen Blattaktoren erreicht. Das stimmt mit dem bekannten Zitat des Erfinders des Aktorsystems Tony Hoare überein:

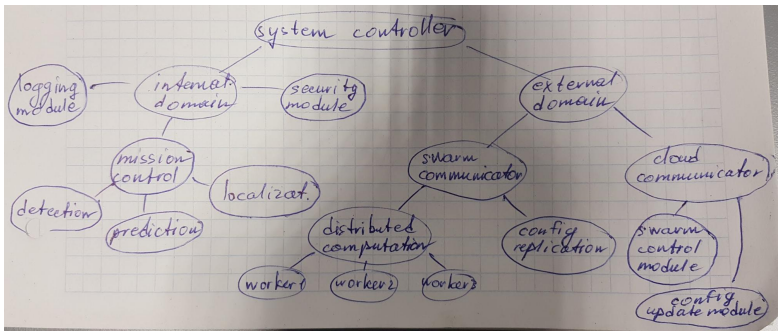


Abbildung 3.3: Möglicher Aufbau eines Systems für Drohnensteuerung mithilfe Aktorsystems.

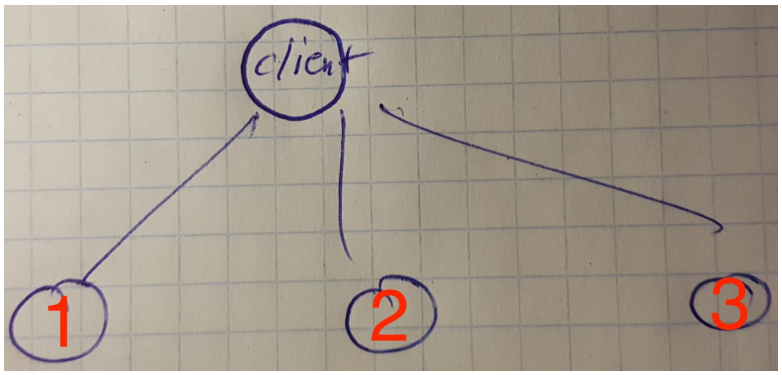


Abbildung 3.4: Möglicher Aufbau eines Systems für Drohnensteuerung mithilfe Aktorsystems.

„There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.“

Das Konzept der Verschiebung des kritischen Codes in äußere Knoten des Baumes nennt man *Error-Kernel* (der Kernel der Fehler).

Bei Aktorsystemen ist es untypisch, wenn eine Hierarchie weniger als zwei oder drei Schichten enthält. Für eine Drohne könnte eine Hierarchie so ausschauen, wie es in der Abbildung 3.3 dargestellt wird.

Der Knoten, der für die Replikation des Zustandes zwischen den Drohnen verantwortlich ist und den Raft-Algorithmus implementiert würde, wäre *Configuration-Replication* (die Replikation der Konfiguration, abgbildet rechts unten). Aufgrund des Aufbaus eines Prototypen war die Hierarchie des Raft-Algorithmus flach gehalten und nur mit zwei Schichten implementiert: eine Schicht des Clients, der Anfragen an das System entgegennimmt, und die andere Schicht der Raft Knoten. Die Hierarchie entspricht der Abbildung 3.4.

Die Knoten 1, 2 und 3 entsprechen den Configuration-Replication-Knoten der ersten

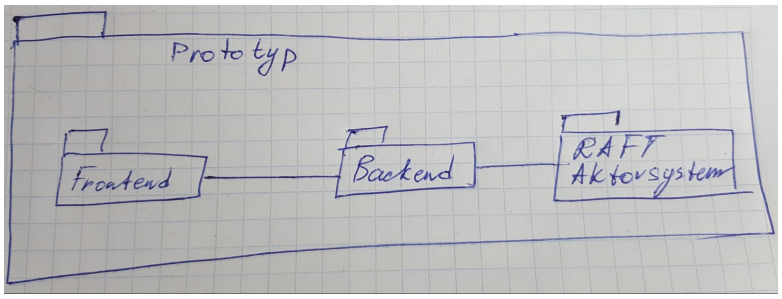


Abbildung 3.5: Teile des Systems.

drei Drohnen. Der Client ist der Knoten, der sich in der Cloud befindet und als Proxy für den Drohnenschwarm dient. Der zu replizierende Wert wird vom Client an einen Knoten geschickt, der die Leader-Rolle im Cluster erfüllt und dann anschließend ohne Beteiligung des Clients laut Raft-Algorithmus zwischen Knoten repliziert.

3.1.4 Annahmen für Server und Frontend des Prototypen

Heutzutage werden vor allem zwei Frameworks für die Entwicklung von Webdiensten auf der JVM verwendet: Jakarta EE und Spring. Scala bietet ein eigenes Framework namens Play. Dies ist für die Entwicklung von MVC- (Model View Controller) und REST-Diensten (Representational State Transfer) geeignet. Play beruht auf Akka-HTTP. Akka-HTTP ist kein Webframework, sondern ein allgemeines Werkzeug in der Akka Werkzeugpalette, um HTTP basierte Dienste aufrufen und HTTP-Anfragen beantworten zu können.

Play bietet folgende Vorteile:

1. standardmäßige Unterstützung von Scala
2. einfache integration mit Akka Aktorsystem

Für die Entwicklung des Frontends wurde React verwendet. React ist ein Framework von Facebook, das eine schnelle (im Vergleich mit reinen JavaScript und HTML) Entwicklung von Webanwendungen ermöglicht.

3.2 Beschreibung der Teilkomponenten und Schnittstellen des Systems

Betrachtet man den entwickelten Prototypen abstrakt, kann er auf drei Bestandteile reduziert werden: das Frontend, das Backend und das Aktorsystem (sehe Abbildung 3.5).

Das Aktorsystem wurde separat dargestellt, um zu betonen, dass das Aktorsystem eine wichtige und stichhaltige Komponente repräsentiert. Während des Entwicklungsprozesses war diese Ansicht auch dadurch gestützt, dass das Aktorsystem in einem separaten

Name	Methode	Pfad	Beschreibung
Zustand des Clusters	GET	/cluster/state	Abfrage des Zustandes eines Raft-Clusters. Die resultierende *JSON-String* enthielt Informationen über einen Leader, die Followers und die Kandidaten. Diese Informationen umfassen unter anderem den Zustand des Logs, den Commit-Index (der Index des zuletzt replizierten Logeintrags) und die Nummer des aktuellen Terms.
Ein neues Element replizieren	POST	/cluster/item	Der Befehl speichert den Wert auf dem Cluster und liefert den aktuellen Zustand des Leader-Logs, wenn der Wert erfolgreich repliziert wurde.

Tabelle 3.1: Endpoints der Frontend-Backend Schnittstelle.

Projekt implementiert wurde. Frontend und Backend wurden ebenfalls in eigenen Projekten implementiert.

Der Prototyp verfügt über mehrere Schnittstellen. Sowohl zwischen den *High-Level-Komponenten* als auch zwischen den *Low-Level-Komponenten*. Zuerst werden die Schnittstellen zwischen den High-Level-Komponenten betrachtet.

3.2.1 Schnittstellen zwischen High-Level-Komponenten

Da es im Prototyp unter den High-Level-Komponenten drei Teile gibt, ergeben sich zwei Schnittstellen:

1. Frontend-Backend Schnittstelle
2. Backend-Aktorsystem Schnittstelle

Die Frontend-Backend Schnittstelle ist eine REST-Schnittstelle. Frontend schickt HTTP-Anfragen an die Backend-Endpoints. Die Endpoints sind in der Tabelle 3.1 aufgelistet.

Die Deklaration der oben erwähnten Endpoints ist in der *routes*-Datei und die Implementierung im *ClusterController* zu finden. Der ClusterController benutzt *ClusterService*, um die Kommunikation zwischen dem Controller und dem Raft-Aktorsystem zu abstrahieren.

Zusätzlich befasst sich der ClusterService mit der Initialisierung des Aktorsystems. Die Initialisierung umfasst folgende Schritte:

1. Erstellung von Promises und Futures
2. Erstellung des Aktorsystems. Dabei wird der ClientActor als Root Aktor erstellt.
3. Absenden der ClientStart-Nachricht an das System

Futures sind Kommunikationsschnittstellen, die Ergebnisse an den ClusterService zurückgeben. Eine naheliegende Lösung wäre, anstatt eines Futures eine Callback-Funktion zu verwenden und alle Ergebnisse in diese Funktion zu übergeben. Dies hätte allerdings dem funktionalen Stil von Scala widersprochen. Deswegen wurde die Entscheidung getroffen, Futures zu verwenden. Die Futures, die in Scala verwendet werden, sind dennoch in ihrer eigenen Art herausfordernd:

Wiederverwendung von Futures. Bei der Planung des Prototypen wurde die Entscheidung getroffen, dass es zu verschwenderisch sei, für jede Nachricht, die an das Aktorsystem geschickt wird, ein neues Promise und ein neues Future zu erstellen.

Timeouts. Wenn der Zustand des Clusters abgefragt sein muss, schickt der ClusterService Anfragen an das Aktorsystem, um den aktuellen Stand des Leader, Followers und Kandidaten zu erhalten. Am Start des Clusters kann es vorkommen, dass der Zustand dann abgefragt wird, wenn es im Cluster z.B. noch keinen Leader oder keine Kandidaten gibt. In dieser Situation wäre die Zeit der Zustellung eines Ergebnisses undefiniert und unbegrenzt.

Um diese Herausforderungen zu meistern, wurden *ReentrantPromise* und *ReentrantFuture* erstellt. Die beiden Namen wurden analog zum *ReentrantLock* in Java ausgewählt, um die Wiederverwendbarkeit der Instanzen dieser Klasse hervorzuheben.

ReentrantPromise setzt die Ergebnisse der asynchronen Operationen und ReentrantFuture erlaubt es, diese Ergebnisse mehrmals abzufragen. Das Ergebnis eines standardmäßigen Futures darf nur einmal aufgerufen werden. Wenn das Ergebnis eines Futures zum zweiten Mal abgefragt wird, wird eine Exception ausgelöst. Dies ist ein Schutzmechanismus, um eine unendliche Blockierung des aufrufenden Threads zu vermeiden, wenn das Ergebnis mehr als einmal abgefragt wird.

Die in Scala definierten Futures erlauben es auch nicht ein Timeout bei der Abfrage des Ergebnisses zu setzen. Eine lange Wartezeit ist für benutzerorientierte Services nicht wünschenswert. Um das zu verhindern, wurde im ReentrantFuture eine Methode mit einem Timeout als Übergabewert zum Abfragen des Ergebnisses hinzugefügt. Wenn innerhalb eines Timeouts kein Ergebnis zur Verfügung steht, wird ein leeres Ergebnis zurückgegeben.

Mittels ReentrantFutures werden folgende Informationen aus dem Aktorsystem geliefert:

1. Zustand des jeweiligen Leaders
2. eine Liste von Zuständen jedes Followers
3. eine Liste von Zuständen jedes Kandidaten
4. Zustand des Leader-Logs, nachdem ein neues Element im Cluster repliziert wurde

Futures werden im Aktorsystem für die Zustellung der Ergebnisse alle Operationen verwendet. Um diese Operationen zu starten, müssen Nachrichten an das System geschickt werden. Beschreibung verwendeter Nachrichten findet man in Tabelle 3.2.

Der Aktor, an den das Ergebnis zugestellt wird, ist immer der Client-Aktor. Dieser setzt das Ergebnis als Ergebnis des jeweiligen Promises. Promise liefert das Ergebnis

Name der Nachricht	Beschreibung	Parameter
ClientStart	Da kein Discovery-System für Aktoren implementiert wurde, startet diese Nachricht ein manuelles Discovery Verfahren.	keine
Leader.Debug.InfoRequest	Den Zustand des Leaders abfragen, sofern es einen gibt.	replyTo - der Aktor, an den das Ergebnis zugestellt sein muss.
Candidate.Debug.InfoRequest	Den Zustand der Kandidaten abfragen.	replyTo - der Aktor, an den das Ergebnis zugestellt sein muss.
Follower.Debug.InfoRequest	Den Zustand der Follower abfragen.	nodeId - Id des Knoten, der Abgefragt sein muss. replyTo - der Aktor, an den das Ergebnis zugestellt sein muss.
ClientRequest	Einen Wert auf dem Cluster replizieren.	value - der zu replizierende Wert. replyTo - der Aktor, an den das Ergebnis zugestellt sein muss.

Tabelle 3.2: Beschreibung der Nachrichten zur Kommunikation mit dem Aktorsystem.

mittels Thread-Synchronisation an Future, sofern das Timeout bei der Abfrage des Ergebnisses noch nicht abgelaufen ist. Das Ergebnis wird vom ClusterService erhalten und an Frontend geschickt.

3.3 Kommunikationsprotokoll zwischen den Teilkomponenten und den Knoten in Raft

Der erste Schritt bei der Implementierung des Raft-Algorithmus hat eine sorgfältige Analyse des Originalartikels erfordert. Obwohl auf Seite 4 des Raft-Artikels eine Zusammenfassung aller Rollen, Nachrichten und Feinheiten der Verhalten dargestellt wird, sind zusätzliche Informationen und Feinheiten im gesamten Artikel zu finden. Im nächsten Schritt waren alle Feinheiten nach den Rollen des Algorithmus gruppiert und in drei Tabellen entsprechend zusammengefasst.

3.3.1 Definition des Followers in Raft

Der Follower ist ein passiver Knoten in Raft und reagiert nur auf die Anfragen des Leaders und der Kandidaten. Jeder Knoten im Cluster startet im Followerzustand und setzt sofort einen Heartbeat-Timer. Wenn der Timer abläuft und sich noch kein einziger Leader gemeldet hat, wird er zum Kandidaten. Der Übergang in eine neue Rolle wird

Name der Nachricht	**Payload**	**Änderungen innerhalb des Knotens**	**Antwort auf diese Nachricht**
AppendEntries Heartbeat des Leaders	- Leader-Information (Term und Akka-Referenz auf den Leader)	- Heartbeat Timer zurücksetzen - wenn der Term des Leaders größer ist als der zuletzt gesehene Term, werden Information über den Leader geupdatet	keine
AppendEntries mit einem neuen Log Eintrag	- Leader-Information - Letzter Logeintrag des Leaders - das zu replezierende Element - Leader-Commit - UUID des Logeintrags (in einer Fußnote erklären)	- wenn der Term des Leader kleiner als der Term des zuletzt gesehenen Leaders ist, wird die Abarbeitung dieser Nachricht abgebrochen. Andernfalls werden weitere Schritte ausgeführt - Leader Information updaten - den letzten Logeintrag des Leaders mit dem eigenen letzten Logeintrag vergleichen. Wenn sie unterschiedlich sind, dann wird der Leader um die Zusendung des vorletzten Eintrags gebeten und weitere Schritte werden nicht ausgeführt. Wenn sie gleich sind, wird der Eintrag an den eigenen	- ein nicht erfolgreicher AppendEntriesResponse, um dem Leader darauf hinzuweisen, dass er diesem Knoten bisherige Logeinträge senden soll - ein erfolgreicher AppendEntriesResponse mit dem UUID des neuen Elements, wenn ein neuer Logeintrag an den lokalen Log angehängt wurde. Ein UUID hilft dem Leader zu identifizieren, welche Einträge und wie vielen Knoten schon erfolgreich repliziert wurden

Tabelle 3.3: Nachrichten, die ein Follower im Raft-Algorithmus empfangen kann.

durch die Verwendung von Akka-Behaviors erreicht. Die Liste von Nachrichten, die ein Knoten in dieser Rolle empfangen kann, ist in der Tabelle 3.3 dargestellt.

3.3.2 Definition des Kandidaten in Raft

Die Übergangsrollen spielen in Raft die Kandidaten. Jeder Follower, dessen Heartbeat-Timer abgelaufen ist, kann zum Kandidaten werden. Der Erfolg des jeweiligen Kandidaten hängt hauptsächlich davon ab, wie kurz sein Heartbeat Timeout ist.

Nach dem Übergang von einem Follower- in einen Kandidatenzustand, startet der Kan-

Name der Nachricht	Payload	Änderungen innerhalb des Knotens	Antwort auf diese Nachricht
VoteGranted	kein	- Die Anzahl von Stimmen um 1 inkrementieren - Checken, ob Stimmen schon vom Quorum aller Knoten empfangen wurden. Wenn es der Fall ist, dann zum Leader werden	keine
AppendEntries	- Leader-Information (Term und Akka-Referenz auf den Leader)	- Wenn der Term des Leader größer oder gleich dem Kandidaten-Term ist, dann zum Leader werden	keine

Tabelle 3.4: Nachrichten, die ein Kandidat empfangen kann.

Name der Nachricht	Payload	Änderungen innerhalb des Knotens	Antwort auf diese Nachricht
RequestVote	- Term des Kandidaten - Akka-Referenz auf den Kandidaten - Der letzte Logeintrag	Am Start jedes Wahlprozesses	An alle Knoten im Cluster unabhängig von ihren Rollen

Tabelle 3.5: Nachrichten, die ein Kandidat aussenden kann.

didat einen Election-Timer (Wahl-Timer). Wenn innerhalb der Election-Zeit der Kandidat nicht zum Leader wird oder kein anderer Leader sich mit einer AppendEntries Nachricht meldet, beginnt die nächste Runde des Wahlprozesses. Alle anderen Veränderungen im Zustand eines Kandidaten werden durch empfangene und gesendete Nachricht eingeleitet, die in Tabellen 3.4 bzw. 3.5 abgebildet sind.

3.3.3 Definition des Leaders in Raft

Der Leader in Raft ist der *Garant des Konsenses* und spielt eine zentrale Rolle. Alle Schreiboperationen erfolgen über den Leader, der dazu noch das Replikationsverfahren steuert und alle Followers am aktuellen Stand hält. Ein Leader existiert innerhalb eines Terms und der Term endet gleich nach dem Absturz des Leaders.

Nachdem ein Kandidat die Wahl gewonnen hat und zum Leader geworden ist, etabliert er sein Leadership durch die erste Heartbeat-Nachricht, die an alle Knoten geschickt wird. Die Heartbeat-Nachricht muss der Leader regelmäßig aussenden, sodass Followers

wissen, dass der Leader noch existiert. Deswegen setzt er einen Timer und schickt die Nachricht alle N Sekunden. Diese Zeit N muss unbedingt kleiner sein als die Zeit des Election-Timers eines Followers. Wenn das nicht der Fall ist, werden Followers immer schneller entscheiden, dass der Leader abgestürzt oder nicht mehr erreichbar ist. Unter diesen Umständen kann ein Leader nie ausgewählt werden.

Der Leader ist eine Schnittstelle zwischen Clients und dem Rest des Clusters. Die Nachrichten, die ein Leader aussendet und erhält, sind in Tabellen 3.6 und 3.7 aufgelistet.

Bei der AppendEntries-Response Nachricht wurde *NextIndex* erwähnt. NextIndex ist ein Map, das Followerreferenzen auf den nächsten Index im Log abbildet. Der jeweilige Follower erwartet diesen Index. Der Leader hält das Map im aktuellen Zustand, um zu wissen, welcher Logeintrag für welchen Follower zu einem Zeitpunkt zugeschickt sein muss, und um sicher zu stellen, dass jeder Log Eintrag genau einmal zugeschickt wird. Es entspricht der *„exactly once“* Nachrichtenzustellungssemantik in verteilten Systemen. Das heißt, die Nachricht darf nicht verloren gehen und darf auch nicht zweimal zugestellt werden.

.
.
.

1.
2.
3.
4.

Name der Nachricht	**Payload**	**Änderungen innerhalb des Knotens**	**Antwort auf diese Nachricht**
ClientRequest	- Der zu replizierende Wert - Die Adresse der Knoten, an den das Ergebnis der Replikation geschickt werden muss	- Einen neuen Eintrag an eigenen Log anhängen - Den neuen Logeintrag auch in die Pending-Item Liste einfügen. Diese Liste enthält Informationen, auf wie vielen Knoten ein Logeintrag schon repliziert wurde. - Eine Append-Entries Nachricht an alle Knoten im Cluster aussenden. So eine Nachricht fungiert auch als ein Heartbeat.	keine
AppendEntriesResponse	- Ein boolescher Wert, der zeigt, ob ein Wert auf einem Knoten erfolgreich / nicht erfolgreich repliziert wurde. - UUID des Logeintrags in der PendingItems Liste - Akka-Referenz auf den Follower, der diese Nachricht geschickt hat	- Wenn ein Wert nicht erfolgreich repliziert wurde, dann ist der Log des Followers immer noch im inkonsistenten Zustand. Man muss 1 von NextIndex subtrahieren und versuchen, den Logeintrag vor dem NextIndex an den Follower zu schicken - Wenn ein Wert erfolgreich repliziert wurde, checken, ob das schon beim ersten Versuch erfolgreich war. In diesem Fall muss UUID des Eintrags in PendingItem in der Nachricht vorhanden sein. - Wenn ein Wert erfolgreich repliziert wurde aber UUID nicht da ist, dann ist es ein Follower, dessen Log immer noch nicht konsistent ist. Follower bittet um Zusendung des nächsten Logeintrags.	- Wenn ein Wert nicht erfolgreich repliziert wurde, dann AppendEntries mit dem vorherigen Logeintrag schicken - Wenn der Wert auf ein Quorum repliziert wurde, Client-Response Nachricht mit dem aktuellen Stand des Zustandsautomaten zurück an den Client schicken - Wenn ein Logeintrag bei einem Follower mit dem inkonsistenten Log erfolgreich repliziert wurde, dann AppendEntries mit dem nächsten Logeintrag schicken
AppendEntriesResponse			keine

Tabelle 3.6: Nachrichten, die ein Kandidat aussenden kann.

Name der Nachricht	**Payload**	**Änderungen innerhalb des Knotens**	**Antwort auf diese Nachricht**
Heartbeat	- Leader-Information - Leader-Commit. Es zeigt den Index des letzten Logeintrags, den Leader commited hat.	Regelmäßig, wenn Leader Timer abläuft	An alle Knoten im Cluster, unabhängig von ihren Rollen

Tabelle 3.7: Nachrichten, die ein Leader aussenden kann.

	Aktormodell	**CSP**	**Funktionale Programmierung**	**Threads und Mutex**
Primär Fokus	Fehlertoleranz und verteiltes Rechnen	Flexibilität und Ausdruckskraft	Nebeneffektefreiheit durch Immutability	Effizienz und breite Anwendbarkeit

Tabelle 3.8: Vergleich von Concurrency Modellen.

Kapitel 4

Fazit

Anhang A

Technische Informationen

Anhang B

Ergänzende Inhalte

Auflistung der ergänzenden Materialien zu dieser Arbeit, die zur digitalen Archivierung an der Hochschule eingereicht wurden (als ZIP-Datei).

B.1 PDF-Dateien

Pfad: /

thesis.pdf . . . . . Finale Master-/Bachelorarbeit (Gesamtdokument)

B.2 Mediendaten

Pfad: /media

*.ai, *.pdf . . . . . Adobe Illustrator-Dateien
*.jpg, *.png . . . . . Rasterbilder
*.mp3 . . . . . Audio-Dateien
*.mp4 . . . . . Video-Dateien

B.3 Online-Quellen (PDF-Kopien)

Pfad: /online-sources

Reliquienschrein-Wikipedia.pdf

Anhang C

Fragebogen

Anhang D

LaTeX-Quellcode

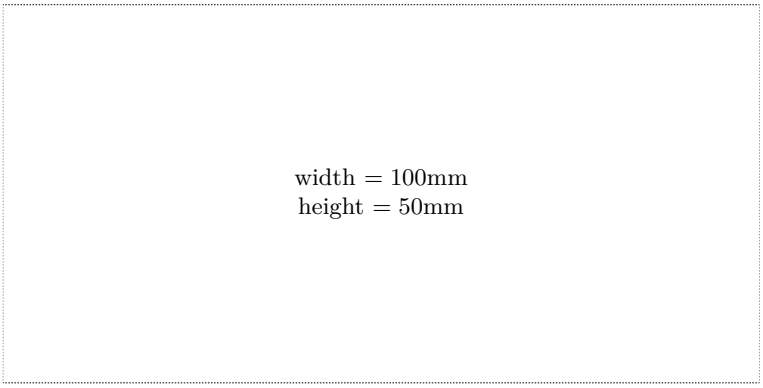
Quellenverzeichnis

Online-Quellen

[1] *Reliquienschrein*. Sep. 2018. URL: https://de.wikipedia.org/wiki/Reliquienschrein (besucht am 28. 02. 2019).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —