Job-Ablaufplanung und Verteilung durch Graphenplanung mit Java

David Riedl

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2017

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.

Hagenberg, am 25. September 2017

David Riedl

Inhaltsverzeichnis

Kurzfassung

Lastenverteilung spielt in modernen Applikationen eine immer größer werdende Rolle, da viele dieser Applikationen durch eine große Multi-User Basis geformt werden, wobei in manchen, diese User miteinander kollaborieren können. Solche Systeme belasten die zugrunde liegende Architektur sehr stark aufgrund der vielen Lasten, die im Hintergrund dafür bearbeitet werden müssen. Es liegt im Interesse der Entwickler, diese Lasten, oder auch Jobs genannt, auf verschiedene Knoten in einem Netzwerk zu verteilen, um die Parallelität zu maximieren. Dabei sind die Größen dieser Jobs ein wichtiger Aspekt, da sie in der Ausführung viel Zeit in Anspruch nehmen können und im Falle eines Fehlers neu ausgeführt werden müssen. Das kann dazu führen, dass gewisse Jobs verhungern und nie ausgeführt werden. Aus diesem Grund ist es wichtig, diese Jobs in kleinere Teile zu partitionieren, um die gesamte Leistung des Systems zu erhöhen. Moderne Frameworks für Java lösen dieses Problem zum Beispiel durch den Einsatz von Aktoren. In dieser Arbeit wird ein weiteres Verfahren vorgestellt, welches alternativ dazu verwendet werden kann, um einen Job in kleinere Teile aufzuspalten, welche im Weiteren parallel ausgeführt werden können. Dabei wird auf das Konzept der Graphenplanung zurückgegriffen, um Abhängigkeiten zwischen den kleinen Teilen zu berechnen. Im ersten Abschnitt wird das fundamentale Problem der Jobgrößen konkret behandelt und im Weiteren der Algorithmus vorgestellt, der im Rahmen des Thesis-Projects umgesetzt wurde. Im zweiten großen Abschnitt der Arbeit werden Strategien, Algorithmen und Notwendigkeiten eines verteilten Systems behandelt, um den Prototypen des Thesis-Projects in ein produktionsfähiges System zu verwandeln.

Abstract

Job or work distribution is becoming a very important and almost necessary part in modern applications as most of them are formed by a multi-user base that, in some cases, collaborates with each other. Such systems place a heavy burden on the architecture they are running on, since a great amount of work has to be done in the background. It is in the interest of the developers to distribute such work in order to maximize parallelism. The size of such jobs is an important aspect, as they can take up a fair amount of processing time and may have to be re-executed on failure, while other jobs end up starving in the sense of concurrency. For that reason, partitioning jobs into smaller portions impacts the overall performance positively. State of the art distribution frameworkes for Java solve this issue by using actor system for example. This thesis introduces an alternative way to divide jobs into smaller parts and schedule them in a parallel way by using graph planning in order to solve inter job dependencies. At first the fundamental problem will be described as well as the basic algorithms needed in order to accomplish said behavior based on the thesis project, while also going into detail about the strategies and necessities of a distributed system, that is able to be used in a production ready environment.

Kapitel 1

Einführung

Moderne Applikationen, welche heutzutage entwickelt werden, haben einen gemeinsamen Trend in Hinsicht auf Parallelität. Dieser existiert nicht nur dahergehend, dass Prozessoren mit immer mehr Kernen ausgestattet werden und damit auch höhere Parallelität unterstützen, sondern auch, weil Netzwerke immer schneller und stabiler werden. Das bedeutet, dass Parallelität nicht mehr auf interne Prozessoren beschränkt ist, sondern auch in verteilten Systemen, in denen mehrere Geräte mittels Netzwerken verbunden sind, denkbar ist. Der Grund, warum Arbeitsverteilung mittels Parallelität ein immer wichtigerer Faktor in Applikationen wird liegt darin, dass Personen heutzutage eine ganz andere Einstellung gegenüber Technik und deren Funktionalität haben.

Immer seltener werden Applikationen, welche einfach installiert werden können und danach nie wieder eine Verbindung mit dem Internet benötigen. Denn eine solche ist heutzutage immer öfter vonnöten, um mit einem dedizierten Server bzw. anderen Benutzern zu kommunizieren. Werden Applikationen herangezogen, welche heutzutage verwendet werden, gibt es meist eine Webapplikation davon. Zum Beispiel hören viele Menschen Musik mittels der Webversion von *Spotify*, vergnügen sich durch das Konsumieren von Videos auf *YouTube* usw. Der große Unterschied liegt darin, dass diese Webapplikationen ihre Daten meist direkt am Server oder in einer Cloud speichern und sichern, oder Inhalte direkt in Zusammenarbeit mit anderen bearbeitet werden können. Ob nun ein Dokument geschrieben, Skizzen gezeichnet oder sogar Software entwickelt wird, alles kann online und gemeinsam mit anderen in Echtzeit erledigt werden. Natürlich dürfen die Gefahren, welche solche Systeme darstellen, nicht unterschätzt werden. Dabei ist das Thema Sicherheit ein wichtiger Aspekt. Diese Themen sind in vielen Fällen transparent für den Endbenutzer, weswegen sie auch gerne vergessen werden und zu dem Phänomen des gläsernen Menschen führen. Diese Themen sind jedoch nicht Inhalt dieser Arbeit. Viel interessanter ist es, sich mit solchen verteilten Systemen auseinander zu setzen und zu verstehen, wie diese funktionieren und welche Konzepte und Prinzipien berücksichtigt werden müssen hinsichtlich des Verteilungsprozesses, um Arbeitslasten mit maximaler Parallelität zu verarbeiten. Um dies zu bewerkstelligen ist es wichtig, den Unterschied zwischen einem Cluster und einem Grid aufzuzeigen, da dies für das Verständnis der restlichen Arbeit von Bedeutung sein wird.

1.1 Cluster

Das Wort Cluster ist in der Informatik sehr stark verbreitet, aber im Grunde bedeutet es immer mehrere, meistens unabhängige, Computer zu einer virtuellen Einheit zusammenzuschließen, indem sie mittels eines Netzwerks verbunden und von einer Software beaufsichtigt werden [24, 26]. Die einzelnen Einheiten in einem Cluster werden als Knoten bezeichnet. Ein triviales Netzwerk von zwei Knoten, welche gemeinsam das selbe Ziel erreichen wollen, kann bereits als solches bezeichnet werden. Cluster werden meist dazu verwendet, um Systeme zu entwickeln, welche, im Gegensatz zu einem einzelnen Computer, hohe Verfügbarkeiten und Performanz aufzeigen sollen. Mit der Entwicklung von immer besseren Werkzeugen zum Implementieren von Clustern und den immer schneller werdenden Netzwerken, geht der Trend bei Online-Applikationen weiter in Richtung verteilter Cluster und weg von der einzelnen Serverstruktur. Dies geschieht nicht nur weil mehrere billige Knoten zusammen die selbe Leistung wie ein teurer Server haben, sondern auch weil solche System mithilfe der *scale-out* Strategie viel skalierbarer sind, nur durch das Hinzufügen von weiteren Knoten. Weil die Knoten in Clustern sehr stark gekoppelt sind, werden sie oft in E-Commerce und Online Banking-System verwendet, in denen Transaktionen, Stabilität, Verfügbarkeit und Skalierbarkeit wichtige Faktoren darstellen.

1.2 Grid

Im Vergleich zu einem Cluster, besteht ein Grid aus komplett voneinander unabhängigen Computern, welche wiederum als Knoten bezeichnet werden. Diese Knoten sind über ein Netzwerk miteinander verbunden. Die Knoten in einem Grid sind viel loser gekoppelt [9, 24]. Auch wenn sie versuchen, zusammen ein gemeinsames Ziel zu erreichen, müssen sie, im Gegensatz zu einem Cluster, nicht zwangsläufig die selbe Arbeit verrichten. Das bedeutet, dass bei einem Grid das Zusammenspiel der einzelnen Knoten nicht direkt mittels einer Software gesteuert werden muss. Das führt im Weiteren dazu, dass die einzelnen Knoten eines Grids viel heterogener und geographisch weiter verteilt sein können. Das bedeutet aber auch, dass die einzelnen Knoten in einem Grid nicht zwangsläufig ähnlich in ihrem hardwarespezifischen Aufbau sein müssen, da sie gänzlich unterschiedliche Arbeiten verrichten können. Selbst wenn sich Cluster und Grids in ihrer Struktur sehr stark unterscheiden können, wirken sie nach außen hin meist wie ein einzelnes Gerät, welches mit einem Interface oder Protokoll angesteuert werden kann.

1.3 Job Distribution

Wird der Fokus in einem Grid-System auf die Verteilung der Arbeitslasten gelegt, wird ein zu bearbeitender Arbeitsschritt oft als Job bezeichnet. Ein Job, in dem Kontext der Arbeitsverteilung, ist eine Beschreibung davon, was erledigt werden muss, welche Ressourcen dafür benötigt werden und was das Endresultat sein wird. Was ein Job im Endeffekt aber widerspiegelt, kann sich von Applikation zu Applikation unterscheiden. Es könnte zum Beispiel eine Transaktion von einem Bankkonto zum nächsten in einer Online Banking-Applikation sein oder etwas ganz simples, wie zwei Benutzer in einem

sozialen Netzwerk als Freunde zu verknüpfen. Diese Jobs können dann in einem verteilten System parallel verarbeitet werden. Dafür gibt es in solchen Systemen meist einige Knoten, welche als Scheduler fungieren und viele Knoten, welche Jobs ausführen und dafür benötigte Ressourcen untereinander aufteilen und verteilen. Wie solch ein System genau funktioniert, wird in Kapitel 5 beschrieben.

Wie bereits erwähnt, spielt das Prinzip der Arbeitslastenverteilung in modernen Applikationen eine immer größer werdende Rolle. Mit dem Trend der Echtzeit Webapplikationen, welche schnell und reaktionsfähig sein sollen, wird es immer wichtiger, große Arbeitslasten im Hintergrund zu verarbeiten, damit es zu keinen Verzögerungen kommen kann. Wie dieses Prinzip auf eine moderne Webapplikation angewandt werden kann, wird im nächsten Abschnitt 1.4 beschrieben.

1.4 Beispiel anhand einer Applikation

Im vergangenen Sommersemester 2016 wurde im Zuge eines Semesterprojektes eine Anwendung entwickelt, anhand welcher viele Aspekte der Lastenverteilung, die in dieser Arbeit beschrieben werden, illustriert werden können. Damit in späteren Kapiteln darauf zurückgegriffen werden kann, wird in diesem Abschnitt diese Applikation genau beschrieben. Die grundlegende Idee ist sehr einfach. Die Benutzer der Applikation haben die Möglichkeit, sich mit einer Vielzahl ihrer sozialen Netzwerke anzumelden. Darunter fallen unter anderem *Facebook*, *Google+*, *YouTube* und *Soundcloud*. Diese werden miteinander verknüpft und die Anwendung beginnt damit, Informationen darüber zu sammeln, welche Handlungen Freunde des eingeloggten Benutzers auf den verschiedenen Netzwerken getroffen haben. Dazu zählt zum Beispiel, welche Videos sie auf *YouTube* angesehen haben oder welche Musik von ihnen auf *Soundcloud* angehört wurde. Diese Aktionen werden durch Einbringen simpler Algorithmen miteinander verbunden, um dem eingeloggten Benutzer anhand der gesammelten Informationen seiner Freunde Vorschläge für Videos und Musik zu geben. Den Namen *SYNCED* erhielt die Applikation aufgrund ihrer eigentlichen Besonderheit, Benutzer miteinander zu synchronisieren. So ist es zum Beispiel möglich, in der Applikation live zu sehen, was sich Freunde gerade anhören bzw. ansehen. Der Benutzer hat dann die Möglichkeit, sich seinem Freund anzuschließen, um das selbe Medium zu konsumieren und springt dabei an die exakt selbe Stelle. In der Terminologie spricht man hierbei von einem Stream. Sobald man einem Stream eines anderen Benutzers beitritt, kann mit diesem gechattet werden. Ein Stream ist jedoch nicht nur auf zwei Personen beschränkt, sondern kann theoretisch unendlich viele Benutzer beinhalten. Daraus ergeben sich einige Anwendungsfälle. So können zum Beispiel Freunde gemeinsam Medien konsumieren, als ob sie im selben Raum sitzen würden, oder ein DJ kann einen Stream öffnen, um seine neuen SoundTracks anderen Benutzern live zu präsentieren. Natürlich ist das Erlangen dieser Informationen kein triviales Problem. Keines der integrierten Netzwerke stellt eine Echtzeitschnittstelle zur Verfügung. Echtzeitschinttstellen sind Schnittstellen, welche nicht wie im herkömmlichen Sinn abgegriffen werden müssen, sondern in Echtzeit Informationen weiterleiten können. Das soziale Netzwerk *Twitter* bietet eine derartige Schnittstelle an. Diese stellt einen Endpunkt zur Verfügung, bei dem sich eine externe Applikation registrieren kann. Diese Applikation muss dann nicht ununterbrochen nach Informationen nachfragen, sondern bekommt diese Informationen direkt zugesendet, wenn auf *Twitter* Aktionen getroffen

werden, welche für die externe Applikation von Bedeutung sind. Für *Facebook*, *Google+*, *YouTube* und *Soundcloud* wurde im Zuge von *SYNCED* ein spezielles Netzwerk entwickelt, welches ständig relevante Informationen bei diesen Diensten abfragt. Eine Abfrage dauert aber je nach Komplexität zwischen ein und fünf Sekunden. Dazu zählt zum Beispiel das Abfragen der Freunde eines Dienstes wie *Facebook* oder das Abrufen verschiedener Playlists wie die Watchhistory von *YouTube*. Damit die Informationen innerhalb von *SYNCED* immer aktuell sind, müssen diese Informationen so oft abgefragt werden wie nur möglich. Um dies zu ermöglichen, wurde ein simples aber effektives Netzwerk von Produzenten und Konsumenten entwickelt. Jeder Produzent bzw. Konsument repräsentiert einen Thread, der am Server läuft. Dabei werden neue Aufgaben, wie zum Beispiel das Abfragen von Informationen, von den Produzenten erzeugt, welche dann von den Konsumenten abgearbeitet werden. Für eine kleine Benutzerbasis hat sich dieses System aufgrund seiner geringen Komplexität bewährt. Jedoch ergeben sich auf lange Sicht gewisse Nachteile. Die Aufgaben, welche produziert werden, wurden als einzelne ausführbare Einheiten entwickelt. Welche Probleme daraus resultieren, wird in Kapitel 2 beschrieben. Außerdem kann ein solches System nicht skalieren, da sich die Parallelität ab einer gewissen Anzahl von Threads nicht mehr steigern lässt. Für eine größere Benutzerbasis muss ein skalierbares System geschaffen werden, welches lediglich mit dem Hinzufügen weiterer Hardware in der Leistung steigt. Wie ein derartiges System entwickelt wurde und welche Herausforderungen dabei bewältigt werden mussten, wird ab dem Kapitel 5 erläutert.

Kapitel 2

Problem der Jobgrößen

Wie bereits erwähnt, wurden in der Beispielapplikation *SYNCED* die verschiedenen Arbeitsblöcke, im weiteren als Job bezeichnet, als einzelne große Jobs entwickelt, welche dann mittels mehrerer Threads abgearbeitet wurden. Dies kann in Lastenverteilungssystemen, in denen hohe Parallelität eine wichtige Rolle spielt, schnell zu Komplikationen führen [13, 16]. Um dieses Problem genauer zu erläutern, wird ein Beispiel aus *SYNCED* herangezogen. Um die verschiedenen sozialen Netzwerke miteinander zu verknüpfen, musste von jedem dieser Netzwerke die Freundesliste eines Benutzers ausgelesen werden, um diese dann zusammenzuführen. Hat ein Benutzer beispielsweise seine Profile für *Facebook*, *Google+* und *Soundcloud* hinterlegt, würden dafür drei Anfragen zu externen Schnittstellen ausgeführt und danach verarbeitet, um die Listen in ein gemeinsames Format zu bringen und zu verbinden. Das sind jedoch bereits vier verschiedene Fehlerquellen, wobei drei davon nicht in der Hand des *SYNCED* Entwicklers liegen, auf Grund des möglichen Versagens der Schnittstellen. Die vierte mögliche Quelle ist das Zusammenführen der einzelnen Freundeslisten. Es könnte passieren, dass ein Datensatz nicht in das einheitliche Format transformiert werden kann, weil sich eine Schnittstelle geändert hat oder ähnliches. Gewisse Fehlerfälle können einfach abgefangen und behandelt werden, aber natürlich nicht alle, angefangen von Netzwerkproblemen, welche mit der Applikation selbst nicht behandelt werden können, bis hin zu technischem Gebrechen. In solchen Situationen muss ein Job erneut ausgeführt werden. Das bedeutet aber auch, dass Anfragen, welche bereits erfolgreich waren, erneut ausgeführt werden. Ein wichtiger Aspekt ist die Zeit, welche in diesem Fall verloren geht, wobei es sich bei externem Abfragen einer Schnittstelle oft um Sekunden handeln kann.

Der einfachste und effizienteste Weg, dieses Problem zu umgehen, liegt darin die Jobs in kleinere Teile aufzuteilen. Diese werden im Weiteren als Tasks bezeichnet, wobei es sich dabei um ausführbare Einheiten handelt und ein Job nun abstrahierter zu sehen ist. Ein Job beschreibt im weitesten Sinn einen Ablauf dieser Tasks, welche parallel ausgeführt werden können bzw. voneinander abhängen. Das wird dazu verwendet, die Reihenfolge festzulegen, um Ergebnisse eines Tasks in einem anderen Task weiter verwerten zu können. Abbildung 2.1 zeigt, wie das Beispiel der Freundeslisten in Tasks aufgeteilt werden kann. Der erste Vorteil, welcher sofort ersichtlich wird, ist das parallele Ausführen der einzelnen Anfragen zu den externen Schnittstellen. Des weiteren müssen nicht alle Abfragen erneut ausgeführt werden, falls eine davon fehlschlägt, sei es nun wegen einem Netzwerkproblem oder einem Ausnahmefall. Eine solche Implementierung

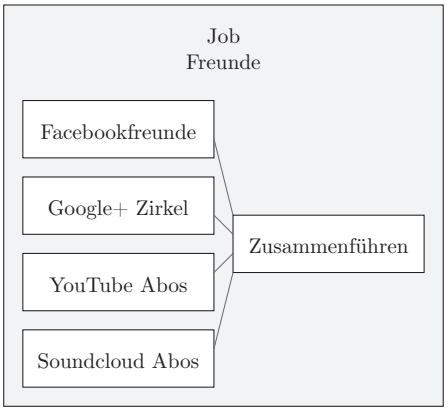


Abbildung 2.1: Aufteilen des Jobs, zum Berechnen einer Freundesliste von verschiedenen sozialen Netzwerken, in kleinere Tasks.

führt schnell zu viel robusteren Systemen die auch viel einfacher zu warten sind, da Ausnahmefälle direkt in den kleinen Tasks behandelt werden können. Außerdem können einfacher andere Strategien im Fehlerfall durchgeführt werden, wie zum Beispiel ein Neustart des gesamten Jobs, eine Neuausführung eines einzelnen Tasks oder ein kompletter Rollback, um etwaige Änderungen rückgängig zu machen.

Eine nicht ganz triviale Herausforderung ist es nun, einen Job in einzelne Tasks aufzuteilen. Dies obliegt dem Programmierer des Systems, doch gibt es dafür verschiedene Herangehensweisen. Im nächsten Abschnitt werden bereits existierende Lösungen präsentiert, um danach einen neuen Ansatz zu erläutern, welcher auf der Graphenplanung basiert.

2.1 State of the Art

Das Problem, welches im vorherigen Abschnitt beschrieben wurde und auf die möglichen Schwachstellen eines verteilten Systems hinsichtlich der Größen der Jobs hinwies, wird bereits von einigen modernen Frameworks für Arbeitsverteilung in verschiedenen Arten gelöst. In diesem Abschnitt werden zwei dieser Frameworks herangezogen, welche die grundlegende Herausforderung der Beispielapplikation aus Abschnitt 1.4 bewerkstelligen und es wird erklärt, welche Ansätze diese verfolgen, um das Problem der Jobgrößen zu lösen.

2.1.1 Quartz

Quartz ist als System dem Prototypen sehr ähnlich, welcher für diese Arbeit implementiert wurde. Man könnte es am ehesten mit einem verteilten CRON vergleichen. Für *Quartz* werden Jobs implementiert, welche analog einem Java Callable sind. Diese Jobs können mittels einem Scheduler abgearbeitet werden. Der Scheduler kann entweder lokal oder verteilt im Netzwerk gestartet und speziell konfiguriert werden, um im Weiteren Jobs abarbeiten zu können. Ein verteiltes System, welches auf *Quartz* aufbaut, ist nicht auf einen Scheduler limitiert. Diese können untereinander kommunizieren und

Lasten verteilen. Der Grund, warum *Quartz* einem CRON sehr ähnelt, kommt davon, dass Jobs mit sogenannten Triggern gekennzeichnet werden können. Diese Trigger definieren, wann und wie oft ein Job in den Schedule aufgenommen werden soll. Auf diesem Wege kann definiert werden, dass ein Job zum Beispiel einmal täglich um Mitternacht, oder auch alle 5 Minuten ausgeführt werden soll.

Um in *Quartz* eine sequentielle Ausführung zweier Jobs zu gewährleisten, muss auf den ersten Job ein Zuhörer gebunden werden, der dem System mitteilt, wann dieser Job fertig abgearbeitet worden ist. Ab diesem Zeitpunkt kann der nächste Job bearbeitet werden. Diese Funktionsweise mag auf den ersten Blick sehr einfach und dynamisch genug sein, um auch in komplexeren Szenarien zum gewünschten Ziel zu kommen. Auf den zweiten Blick wird aber schnell klar, dass zum Beispiel Fälle wie in Abschnitt 1.4 beschrieben zwar möglich sind, aber die Konfiguration dafür recht schnell unübersichtlich und schwer zu warten wird. Das bedeutet im Weiteren auch, dass *Quartz* als grundlegendes Framework nicht geeignet ist, um das Problem der Jobgrößen zu lösen.

2.1.2 AKKA

Im Gegensatz zu *Quartz* verwendet das *AKKA* Framework sogenannte Aktoren anstatt den herkömmlichen Jobs, um Arbeiten zu verteilen. Ein Aktor kann im weitesten Sinn mit einem Job verglichen werden, da er im Grunde eine ausführbare Einheit in einem Aktoren-System ist. Jedoch bietet er weitere Funktionalität, welche ein derartiges System sehr flexibel macht.

Aktoren sind in der Lage neue Subaktoren zu erstellen, wenn zum Beispiel die Arbeitslast zu groß für einen Aktoren alleine ist. Das liegt in der Hand des Entwicklers und geschieht nicht automatisch. Die Arbeit wird sozusagen in kleinere Teile zerlegt, welche jeweils von einem Aktor bearbeitet werden. Dabei übernimmt der ursprüngliche Aktor, welcher die anderen erstellt hat, die Rolle des Betreuers. Dieser Betreuer entscheidet in gewissen Situationen das Vorgehen, welches angewendet werden soll. So liegt es beispielsweise in seiner Verantwortung zu entscheiden, was in Fehlersituationen seiner Subaktoren passieren soll. Diese Verantwortung kann er aber auch an seinen Betreuer weiter delegieren, sofern vorhanden.

Ein derartig hierarchisches System bietet in vielen Fällen große Vorteile. So sind Fehlerquellen schnell auffindbar, da es sehr einfach ist, herauszufinden, welcher Aktor den Fehler verursacht hat und welchen Arbeitsteil er übernehmen hätte sollen. Außerdem können für vorhersehbare Fehler Gegenmaßnahmen im Betreuer definiert werden, um ein System zu entwickeln, welches sich selbst heilen kann.

Damit dieses Prinzip funktionieren kann, müssen Aktoren in der Lage sein, miteinander zu kommunizieren. Dies geschieht mittels Nachrichten, welche Aktoren austauschen können. Das System der Nachrichten wurde aus einem guten Grund gewählt, da dieses soweit abstrahiert werden kann, um in verschiedenen Umgebungen verwendet werden zu können. So muss ein Aktor nicht wissen, wie er mit einem anderen kommuniziert, sei es lokal über Interprozesskommunikation oder über ein LAN bzw. WAN Netzwerk.

Kapitel 3

Jobgrößen und Planungsgraphen

Im vorherigen Kapitel wurde anhand von modernen Frameworks aufgezeigt, welche Techniken und Möglichkeiten bereits vorhanden sind, um das fundamentale Problem der Jobgrößen aus Kapitel 2 zu lösen bzw. zu umgehen. Dieses Kapitel geht auf das grundlegende Konzept ein, welches im Rahmen des Prototypen implementiert wurde, um das Problem der Jobgrößen mithilfe von Planungsgraphen zu lösen.

3.1 Graphenplaner

Graphenplaner sind allgemeine Planer[1], welche mit Hilfe eines sogenannten Planungsgraphen versuchen, einen Weg von einem Startzustand zu einem Zielzustand zu finden. Im Grunde funktionieren sie sehr ähnlich wie ein Suchalgorithmus, was im Laufe dieses Abschnittes verdeutlicht wird. Im Gegensatz zu anderen Suchverfahren wird nicht sofort mit der Suche gestartet, sondern zuerst ein Planungsgraph erstellt. Dieser zergliedert das Planungsproblem in nötige Teilschritte, sodass verschiedene Bedingungen schon sehr früh erkannt werden können. Das bedeutet zugleich, dass nicht der ganze Problemraum durchsucht werden muss, sondern immer nur ein kleiner Teil davon. Planungsgraphen können in polynomialer Zeit aufgebaut werden und bleiben auch bei komplexen Problemen sehr kompakt [2]. Das bedeutet, dass Planungsgraphen nicht mit Zustandsgraphen [14] zu verwechseln sind, welche sehr schnell sehr groß werden können. Ein Plan in einem Zustandsgraphen ist im Grunde ein kompletter Pfad durch den Graphen, während es bei einem Planungsgraph eher ein Ablaufstrom ist, der die Reihenfolge der Aktionen definiert.

Wenn von einem Graphenplaner gesprochen wird, ist damit meist der Algorithmus *GraphPlan* von Blum und Furst [2] gemeint. Dieser hat sich bis heute in dem Gebiet stark durchgesetzt und wurde durch neue Heuristiken [3] angepasst und verbessert. *GraphPlan* verwendet Planungsgraphen, um einen Plan für ein Planungsproblem zu finden und kombiniert dabei Aspekte von total und partiell geordneten Planern. So werden zum Beispiel, wie bei total geordneten Planern, Aktionen mit konkreten Zeitstempeln verbunden, an denen sie ausgeführt werden sollen. Jedoch werden zudem auch partiell geordnete Pläne erstellt. Diese können Aktionen und Schritte parallel ausführen und

[1] Allgemeine Planer sind nicht für ein spezielles Szenario entwickelt, sondern können in vielen verschiedenen Situationen verwendet werden.

sind demnach nur partiell geordnet. In dem Transportproblem von Programm 3.1, können beispielsweise alle Frachten im ersten Zeitschritt eingeladen werden. In Zeitschritt 2 würde das Flugzeug von A nach B fliegen und in Zeitschritt 3 würden alle Frachten ausgeladen werden können.

Graphenplaner sind nur in STRIPS ähnlichen Definitionsbereichen [8] anwendbar. Diese Bereiche definieren Aktionen, welche ausgeführt werden können und eine gewisse Struktur aufweisen. Basierend auf bestimmten Voraussetzungen haben sie positive bzw. negative Auswirkungen. Das bedeutet, es können Auswirkungen sein, mit denen etwas hinzugefügt oder gelöscht werden kann. Dabei handelt es sich immer um Propositionen, Wahrheitswerte, welche mit Konjunktionen verknüpft werden und den derzeitigen Status repräsentieren. Folglich können Aktionen aus einem Status Propositionen entfernen oder neue hinzufügen, wobei eine Aktion nur ausgeführt werden kann, wenn alle ihre Voraussetzungen im derzeitigen Status enthalten sind. Durch das Ausführen von Aktionen ergibt sich wiederum ein neuer Status, auf welchen neue Aktionen ausgeführt werden können. Dabei wird angenommen, dass Propositionen, welche nicht Teil der Auswirkungen einer Aktion sind, unverändert bleiben. Das trägt dazu bei, das Rahmenproblem zu lösen, indem für jede unveränderte Proposition eine Rahmenaktion ausgeführt wird, welches die Proposition von einem Zeitschritt t in den nächsten $t + 1$ kopiert. Zudem können die Parameter der jeweiligen Aktionen mit Objekten instanziert werden und bilden so eine voll instanzierte Aktion. Zum Beispiel könnte die Aktion Fliegen(f, von, nach) so instanziert werden, dass sie Fliegen(Boeing-747, München, Salzburg) ergibt. Zusammengefasst bedeutet das, dass sich das spezifische Planungsproblem zusammensetzt aus

- einem STRIPS ähnlichen Definitionsbereich,
- einer Liste von Objekten,
- einer Liste von Propositionen, die den Startzustand beschreiben und
- einer Liste von Propositionen, die den Endzustand beschreiben.

3.1.1 Struktur eines Planungsgraphen

Bevor die Struktur eines Planungsgraphen erläutert werden kann, ist es wichtig zu definieren, was ein valider Plan ist, da sie sich sehr ähnlich sind, jedoch feine Unterschiede aufweisen. Ein valider Plan im Kontext der Graphenplanung ist eine Abfolge von Aktionen, wobei jede mit einem gewissen Zeitstempel versehen wird. So werden zum Zeitstempel 1 Aktionen ausgeführt, zum Zeitstempel 2 und so weiter. Es können jedoch, wie bereits erwähnt, mehrere Aktionen gleichzeitig ausgeführt werden, sofern diese nicht miteinander kollidieren. Zwei Aktionen kollidieren genau dann miteinander, wenn eine Aktion die Voraussetzungen einer anderen löscht. Ein Planungsgraph unterscheidet sich von einem validen Plan dadurch, dass zusätzliche Informationen vorhanden sind. So werden zu einem Zeitstempel auch Aktionen zugelassen, die miteinander kollidieren. Dabei alterniert ein Planungsgraph immer zwischen zwei Ebenen als gerichteter Graph. Dies sind die Propositionsebene, bei der jeder Knoten jeweils eine Proposition repräsentiert und die Aktionsebene, die dasselbe nur für Aktionen darstellt. Die erste Ebene eines Planungsgraphen ist immer eine Propositionsebene, die alle Propositionen des Startzustandes als Knoten beinhaltet. Die Kanten in einem Planungsgraphen zeigen

Programm 3.1: Definitionsbereich zum Transportieren von Frachten zwischen Flughäfen mittels Flugzeugen.

```
AKTION(Fliegen, {
  Parameter: (
    Flugzeug(f) ∧ Flughafen(von) ∧ Flughafen(nach)
  ),
  Voraussetzungen: (
    NichtGleich(von, nach) ∧ BefindetSichAuf(f, von) ∧ Flugbereit(f)
  ),
  Effekte: (
    ¬BefindetSichAuf(f, von) ∧ BefindetSichAuf(f, nach) ∧ ¬Flugbereit(f)
  )
});

AKTION(Ausladen,
  Parameter: (
    Flugzeug(f) ∧ Flughafen(fh) ∧ Fracht(fr)
  ),
  Voraussetzungen: (
    BefindetSichAuf(f, fh) ∧ BefindetSichIn(fr, f)
  ),
  Effekte: (
    ¬BefindetSichIn(fr, f) ∧ BefindetSichAuf(fr, fh)
  )
);

AKTION(Beladen,
  Parameter: (
    Flugzeug(f) ∧ Flughafen(fh) ∧ Fracht(fr)
  ),
  Voraussetzungen: (
    BefindetSichAuf(f, fh) ∧ BefindetSichAuf(fr, fh)
  ),
  Effekte(
    ¬BefindetSichAuf(fr, fh) ∧ BefindetSichIn(fr, f)
  )
);
```

die Beziehungen zwischen den einzelnen Ebenen auf. So werden zum Beispiel die Knoten einer Aktionsebene mit den Propositionen aus der vorherigen Ebene verbunden, welche ihre Voraussetzungen sind. Zudem wird eine Aktion auch mit den Propositionen in der nächsten Ebene verbunden, die diese Aktion entweder hinzufügt oder löscht. In den Beispielen dieser Arbeit, wird das Löschen von Propositionen dadurch gekennzeichnet, dass deren Negation hinzugefügt wird. Das vereinfacht im Späteren das Verständnis des gegenseitigen Ausschließens. Wie bereits erwähnt, gibt es für jede Proposition eine Rahmenaktion, welche diese in die nächste Ebene kopiert.

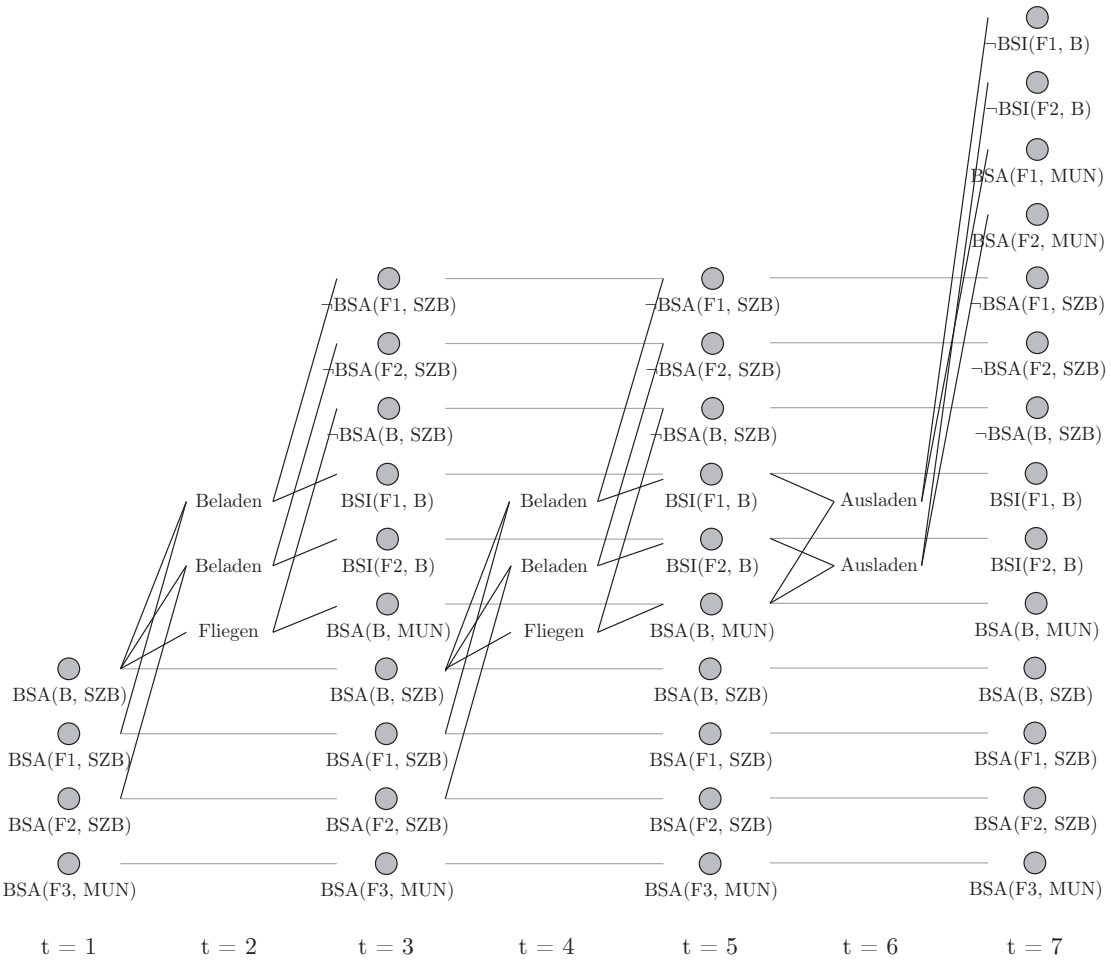


Abbildung 3.1: Planungsgraph, der die einzelnen Schritte anhand des Transportsproblemes demonstriert. Dabei sollen die Frachten *F1* und *F2* von Salzburg (SZB) nach München (MUN) mit dem Flugzeug *B* transportiert werden. Der Planungsgraph in diesem Beispiel weicht, zur Vereinfacherung, etwas vom Algorithmus ab. Ansonsten würden die Auslade-Aktionen bereits in Zeitschritt 4 berücksichtigt werden. Die hellgrauen Verbindungen zweier Propositionen repräsentieren die Rahmenaktionen.

3.1.2 Lösungsweg

Nach jedem Konstruieren einer neuen Ebene, kontrolliert der *GraphPlan* Algorithmus, ob bereits ein Lösungsweg vorhanden ist. Dazu muss definiert werden, was eine Lösung für ein Planungsproblem innerhalb des Planungsgraphen genau ist. Grundlegend ist es ein Subgraph, wobei alle Propositionen des Zielzustandes in einer Propositionsebene vorhanden sein müssen. Außerdem dürfen diese Propositionen, sowie alle Aktionen die zu diesem Zustand führen, sich nicht in ihrer jeweiligen Ebene gegenseitig ausschließen. Dafür gibt es ein Verfahren mit dem festgestellt werden kann, ob Propositionen bzw. Aktionen sich gegenseitig ausschließen und einen sogenannten Mutex bilden. Bei Propositionen ist dies der Fall, wenn eine die Negation der anderen ist (Abb. 3.2(a)) oder alle Aktionen, die diese Propositionen erzeugen, paarweise einen Mutex bilden (Abb. 3.2(b)).

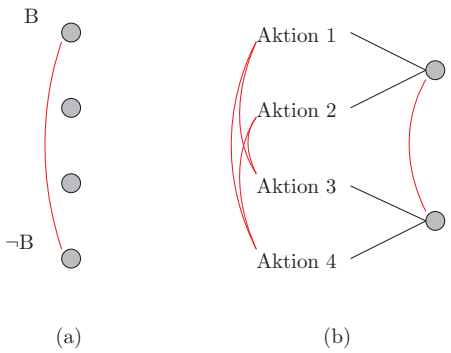


Abbildung 3.2: Zwei Propositionen derselben Ebene können sich gegenseitig ausschließen, wenn sie entweder Negationen voneinander sind oder alle Aktionen, die diese Propositionen erzeugen, sich paarweise ausschalten.

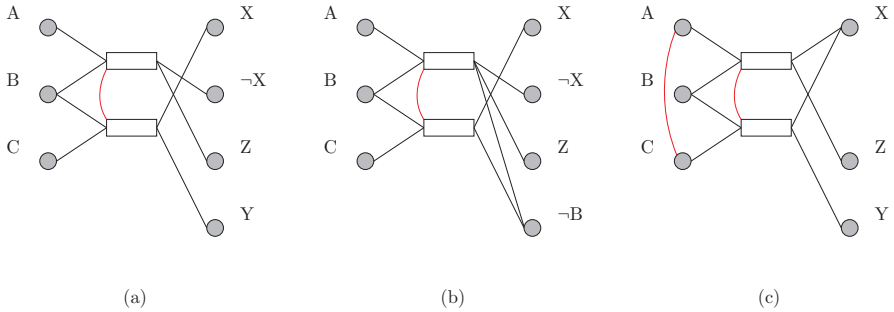


Abbildung 3.3: Zwei Aktionen derselben Ebene können sich gegenseitig ausschließen, wenn sie inkonsistente Effekte haben, ihre Voraussetzungen löschen oder bereits Propositionen als Voraussetzungen haben, die sich gegenseitig ausschließen.

Zwei Aktionen derselben Ebene bilden genau dann einen Mutex, wenn sie inkonsistente Effekte haben. Das bedeutet, dass mindestens ein Effekt einer Aktion die Negation eines Effektes der anderen ist (Abb. 3.3(a)). Wenn eine Aktion die Voraussetzung einer anderen Aktion löscht, bilden diese ebenfalls einen Mutex (Abb. 3.3(b)). Zusätzlich können Aktionen einen Mutex bilden, wenn mindestens ein Paar ihrer Voraussetzungen bereits ein Mutex ist (Abb. 3.3(c)). Zusammengefasst alterniert der *GraphPlan* Algorithmus zwischen zwei Phasen. Zuerst wird der Planungsgraph erstellt, bis eine Propositionsebene erreicht wird, die den Zielzustand beinhaltet und die benötigten Propositionen des Zielzustandes keine Mutexe bilden. Danach wird versucht, einen Lösungsweg zu finden. Dabei arbeitet sich der Algorithmus nun wieder Ebene für Ebene zurück und prüft, ob notwendige Propositionen und Aktionen keine Mutexe bilden. Gelingt es, bis zur ersten Propositionsebene zurückzukommen, wird auf diese Weise ein Subgraph erstellt, welcher gleichzeitig den Lösungsweg repräsentiert. Bilden während dieses Prozesses entweder Propositionen oder Aktionen einen Mutex, wird der Planungsgraph erweitert und das Vorgehen wiederholt sich. Damit sich dieser Prozess nicht endlos wiederholen kann, merkt sich der Algorithmus, welche Subgraphen nicht lösbar sind und bricht früh genug ab, sobald keine neuen Situationen erreicht werden können [2]. Abbildung 3.4 zeigt

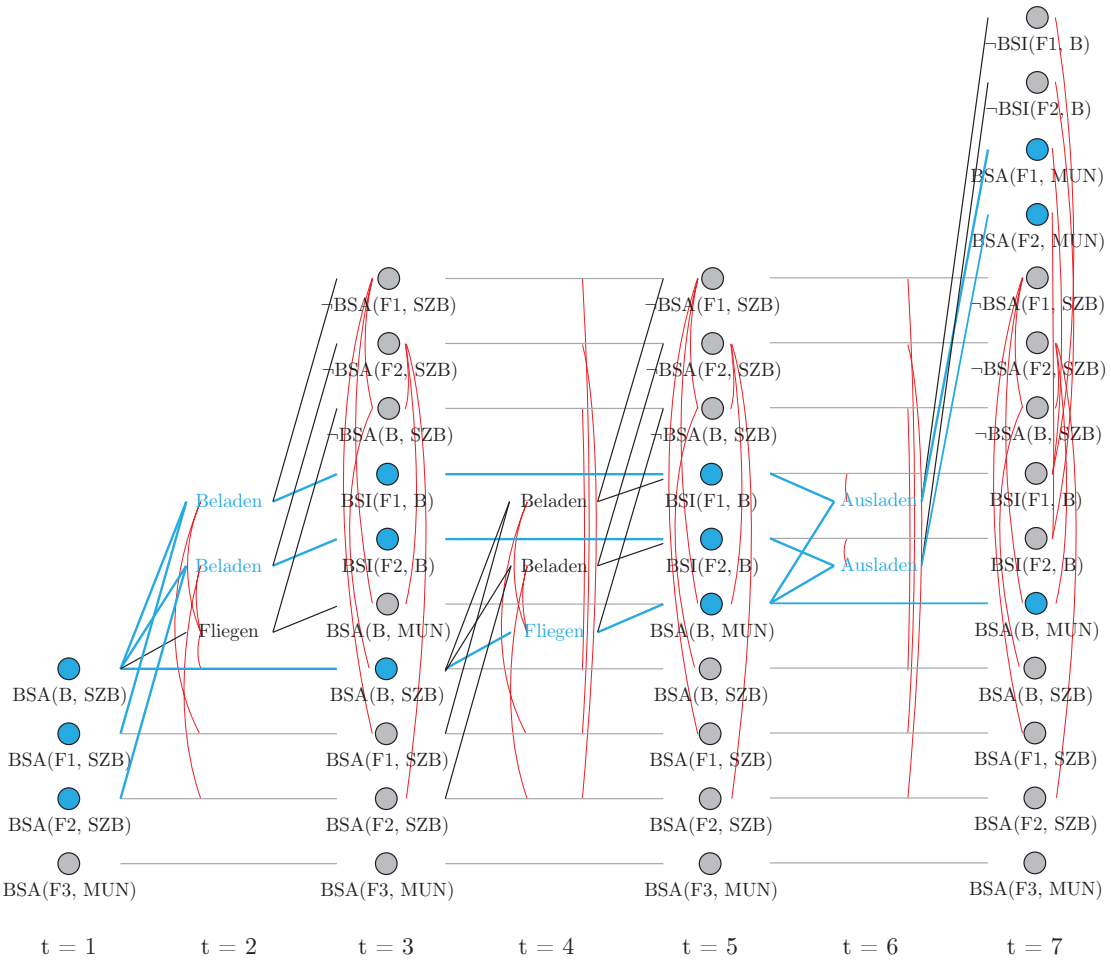


Abbildung 3.4: Ein Mutex zwischen Propositionen bzw. Aktionen wird durch eine rote Verbindung symbolisiert. Der Lösungsweg, welcher von hinten nach vorne gefunden wird, ist blau gekennzeichnet.

am Transportproblem, welche Propositionen und Aktionen sich gegenseitig ausschließen und wie der Lösungsweg zustande kommt. Das Vorgehen unter Verwendung der Mutexe verkleinert den eigentlichen Suchraum und ermöglicht es, unnötige Suchschritte zu verhindern.

3.2 Jobgrößen mit Planungsgraphen

Mit der Graphenplanung kann zu einem Problem ein Lösungsweg gefunden werden, welcher ein partiell geordneter Plan ist. Dieser definiert Aktionen, welche ausgeführt werden müssen, um von einem Ausgangszustand zum Ziel zu gelangen, mit der zusätzlichen Information, welche davon parallel abgewickelt werden können. Im Grunde liefert die Graphenplanung bereits viele Aspekte, welche in einem verteilten Lastenverteilungssystem benötigt werden, wenn es um die Planung der Ausführung geht. Der zweite Aspekt, der bereits von der Graphenplanung automatisch gelöst wird, ist auch gleichzeitig das

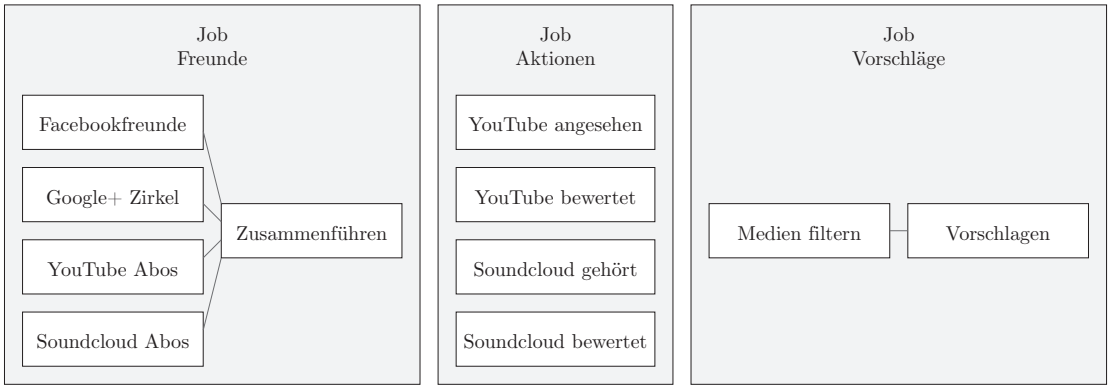


Abbildung 3.5: Diese Abbildung repräsentiert die drei verschiedenen Jobs, um einen Benutzer in *SYNCED* Vorschläge aufzubereiten und auch die elf einzelnen Tasks, in welche man sie aufspalten könnte.

ursprüngliche Problem der Jobgrößen, welches bereits behandelt wurde. Dieses Problem lässt sich recht einfach an einem weiteren Beispiel aus der Applikation *SYNCED* veranschaulichen. Wie bereits erwähnt, gab es in *SYNCED* die Möglichkeit, einem Benutzer Vorschläge für Medien aufzubereiten, anhand der Aktionen, welche seine Freunde auf anderen Netzwerken getroffen haben. Ausgegangen von einem Benutzer, welcher *Facebook*, *Google+*, *YouTube* und *Soundcloud* in *SYNCED* eingebunden hat, muss als erstes seine Freundesliste aktualisiert werden. Des weiteren werden von YouTube die Videos abgefragt, welche von seinen Freunden angesehen, positiv bewertet oder kommentiert wurden. Selbiges gilt für Soundcloud, um auch musikalische Vorschläge aufzeigen zu können. Nachdem diese Informationen gesammelt sind, werden sie anschließend gefiltert, weil nicht alle Aktionen von Freunden relevant für einen Benutzer sind, sondern nur solche, welche mit den eigenen Interessen zu einem gewissen Grad übereinstimmen. In *SYNCED* wurde dies mittels drei verschiedener Jobs gelöst. Ein Job löst die Freundesliste auf, der zweite fragt die Aktionen auf *YouTube* und *Soundcloud* ab und der dritte filtert die Resultate und gibt dem Benutzer Vorschläge für Medien. Abbildung 3.5 illustriert diese drei Jobs und gleichzeitig auch die einzelnen Tasks, in welche man sie aufspalten könnte, um eine höhere Effizienz und Robustheit zu erreichen. Wenn in einer Programmiersprache wie Java ein derartiger Task implementiert wird, verwendet man dafür meistens eine Klasse, die einem Runnable sehr ähnlich ist. Das sind Klassen mit einer speziellen Methode, welche die Logik des Tasks beinhaltet. Dadurch können Tasks in einem einzelnen Thread oder sogar verteilt auf einer anderen Maschine ausgeführt werden. Zudem braucht ein Task meist Parameter, mit denen er arbeitet und produziert dabei ein Ergebnis, welches weiter verarbeitet wird, oder bereits das Endresultat ist. Im Grunde funktioniert dieses System also sehr ähnlich wie ein Planungsgraph. Es gibt verschiedenste Aktionen mit Voraussetzungen und Auswirkungen, in diesem Fall Ergebnissen. Dies bedeutet aber auch, dass einem verteilten System, welches mit der Graphenplanung arbeitet, lediglich ein Startzustand und ein Zielzustand gegeben werden muss. Dadurch kann der gesamte Job berechnet werden, der wiederum definiert, in welcher Reihenfolge die einzelnen Tasks ausgeführt werden müssen und welche davon parallel abgearbeitet werden können.

Programm 3.2: Abstraktes Grundgerüst eines Tasks, welches vererbt werden kann, um einen Task zu implementieren.

```
1  public abstract class AbstractTask implements Runnable, Serializable {
2    private Throwable reason;
3
4    @Override
5    public void run() {
6      try {
7        this.task();
8      } catch(Throwable exception) {
9        this.reason = exception;
10       this.failed();
11     }
12
13     this.finished();
14   }
15
16   public abstract void task();
17 }
```

3.3 Implementierung

In diesem Abschnitt wird auf die konkrete Umsetzung des Systems eingegangen, welches es ermöglicht, mittels der Graphenplanung, anhand eines Startzustands und eines gewünschten Endzustands, einen Job zu berechnen. In einem solchen System bildet ein Task die kleinste Einheit. Dieser repräsentiert im Graphenplan genau eine Aktion und im verteilten System eine ausführbare Einheit. Programm 3.2 zeigt auf, wie das Grundgerüst eines solchen Tasks implementiert werden kann. Diese Klasse implementiert das Interface Runnable, damit sie im Weiteren ausgeführt werden kann. Wie das Ausführen eines solchen Tasks ablaufen kann, wird in Kapitel 5 beschrieben. Außerdem ist es wichtig, das Interface Serializable zu implementieren, damit der Task zwischen verschiedenen Knoten im Netzwerk verschickt werden kann, um die Lasten zu verteilen. Die konkrete Logik des Tasks wird in die Methode `task()` ausgelagert, damit in der `run()` Methode noch auf gewisse Fehler eingegangen werden kann. Wirft zum Beispiel der Task in seiner Logik eine Exception, wird diese abgefangen, intern als Grund des Versagens gesetzt und mit der `failed()` Methode signalisiert, dass es zu einem Fehler gekommen ist. Das Gegenstück dazu ist die `finished()` Methode, welche signalisiert, dass der Task erfolgreich ausgeführt wurde. Wie diese Signale behandelt werden, wird vorerst nicht genauer erläutert. Die Frage, die sich in diesem Kontext stellt, ist, wie man einen solchen Task mit dem Algorithmus der Graphenplanung verknüpfen kann. Wie zuvor erwähnt, haben Aktionen in einem Planungsgraphen Voraussetzungen und Auswirkungen auf einen Status. Das bedeutet, dass es eine Möglichkeit geben muss einen Task zu konfigurieren, um definieren zu können, welche Propositionen er benötigt und wie er sich auf den derzeitigen Status auswirkt. Propositionen können recht einfach mit sogenannten Java Beans realisiert werden. Das sind Klassen, die als Datenhalter dienen und für gewisse interne Variablen Getter- und Setter-Methoden anbieten, oft zusammengefasst als Accessor-Methoden bezeichnet. Benötigt zum Beispiel ein Task eine

Programm 3.3: Annotationen für den Task zum Abfragen von Facebook-Freunden eines Benutzers.

```
1  @Preconditions({
2    UserId.class
3  })
4  @Effects({
5    FacebookFriendList.class
6  })
7  public class FetchFacebookFriendsTask extends AbstractTask {
8    @Override
9    public void task() { ... }
10 }
```

Benutzer-Identifikationsnummer (ID), kann dafür ein Bean erstellt werden (`UserId`), der die Accessor-Methoden `getId(), setId()` zur Verfügung stellt. Um zu definieren, dass ein Task eine `UserId` als Voraussetzung hat, werden Java-Annotationen verwendet. Mittels Annotationen können in Java Klassen, Methoden, Instanzvariablen, Konstruktoren und mehr genauer beschrieben und erweitert werden. Um genau zu sein, wurden im Prototypen vier Annotationen für die Graphenplanung entwickelt. Mit der Annotation `@Preconditions` kann direkt für einen Task definiert werden, welche Beans als Voraussetzungen benötigt werden. Die Annotationen `@Effects` und `@NegativeEffects` können dazu verwendet werden, um zu beschreiben, welche Beans dieser Task erstellt und zum Status hinzufügt oder konsumiert und aus dem Status entsprechend entfernt. Die vierte Annotation `@Action` legt fest, ob ein Task in der Graphenplanung berücksichtigt werden soll, was später noch genauer erklärt wird. Programm 3.3 zeigt auf, wie zum Beispiel der Task, welcher die Freunde für einen Benutzer von *Facebook* abruft, annotiert werden kann. Er benötigt dazu die ID des Benutzers, dessen Freunde abgefragt werden sollen und liefert ausgehend davon eine Liste von Facebookfreunden zurück.

Wie zuvor erwähnt, werden alle Tasks, die in der Graphenplanung berücksichtigt werden sollen, mit der Annotation `@Action` gekennzeichnet. Dadurch ist es selbst zur Lauftzeit recht einfach durch Reflections[2] alle Klassen zu finden, welche einen Task repräsentieren. Jede Klasse, die mit der Annotation `@Action` gekennzeichnet ist und von der Klasse AbstractTask erbt, ist für die Graphenplanung zulässig. Diese Aktionen werden zur Laufzeit in einen Behälter geladen, welcher alle validen Aktionen für die Graphenplanung sammelt. Im Abschnitt 3.1.1 wurde erwähnt, dass der *GraphPlan* Algorithmus das Rahmenproblem mittels sogenannter Rahmenaktionen löst, welche Propositionen von einer Propositionsebene in die nächste kopieren. Dafür muss für jede Proposition, welche in den Voraussetzungen bzw. Auswirkungen der gefunden Aktionen vorkommt, eine Rahmenaktion in den Behälter hinzugefügt werden. Rahmenaktionen haben als Voraussetzung und gleichzeitig auch als Auswirkung die Proposition, für die sie erstellt wurden. Dieses Vorgehen muss im Weiteren berücksichtigt werden, da eine

[2] *Reflections* in Java ist ein Mechanismus, der es dem Programm ermöglicht Code desselben Systems zu inspizieren und zu verstehen, um beispielsweise alle Methoden oder Instanzvariablen einer Klasse auszulesen oder wie in diesem Fall, um alle Klassen zu finden, die mit einer gewissen Annotation gekennzeichnet sind.

Rahmenaktion keinen Task repräsentiert, sondern lediglich für die Graphenplanung benötigt wird. Aus diesem Grund werden Aktionen in die folgenden Kategorien eingeteilt:

- Task-Aktionen, welche jeweils einen konkreten Task repräsentieren und später ausgeführt werden können.
- Rahmenaktionen, welche rein für die Graphenplanung verwendet werden und für die Ausführung und Lastenverteilung nicht relevant sind.

Der erstellte Aktionenbehälter ist nach dem Sammeln aller Aktionen in der Lage, für einen gegebenen Status eine Liste von Aktionen zu berechnen, welche auf diesen Status angewendet werden können. Dabei werden beide Kategorien von Aktionen berücksichtigt. Ausgehend von einem Startzustand ist es nun recht einfach, einen Graphenplan zu erstellen. Der Plan ist durch eine doppelt verkettete Liste gelöst, wobei jeder Eintrag in der Liste eine Ebene im Graphen repräsentiert. Außerdem ist jede Ebene in der Lage, die nächste Ebene zu erstellen, da sie alle nötigen Informationen dafür gespeichert hat. Eine Propositionsebene speichert den derzeitigen Status in Form einer Liste von Propostionen und kann mithilfe des Aktionenbehälters eine neue Aktionsebene erstellen. Anhand der Auswirkungen aller Aktionen einer Aktionsebene, kann wiederum eine Propositionsebene generiert werden. Programm 3.4 zeigt auf, wie dieser Algorithmus abstrakt funktioniert. Dieser wiederholt sich so lange bis er gelöst wurde, dabei muss jedoch nicht zwangsläufig eine Lösung gefunden werden. Bevor die nächste Ebene erstellt wird, durchläuft der Algorithmus eine Routine. Diese berechnet für eine Ebene die derzeitigen Mutexe und prüft, sofern es eine Propotionsebene ist, ob bereits der Zielzustand erreicht ist. Ist das der Fall, wird versucht eine Lösung zu finden. Dabei kann es vorkommen, dass zu diesem Zeitpunkt noch keine Lösung gefunden werden kann. Wie in Abschnitt 3.1.2 beschrieben, wird eine nicht vorhandene Lösung nicht als Abbruchbedingung gewertet, sondern das wiederholte Besuchen einer Ebene. Deshalb wird in der Routine auch gespeichert, welche Ebenen bereits besucht worden sind. Wird diese Abbruchbedingung erreicht, ist das Problem gelöst, es wurde jedoch keine valide Lösung gefunden.

Die zwei unterschiedlichen Ebenen für Propositionen und Aktionen wurden mittels zweier Klassen gelöst, die beide in der Lage sind, Mutexe zu berechnen. Dies wird dadurch gewährleistet, dass die Ebenen eine doppelt verkettete Liste bilden. Das bedeutet, dass eine Ebene zum Zeitstempel t eine Referenz auf die Ebenen mit den Zeitstempeln $t-1$ und $t+1$ hat. Laut der Methode aus Abschnitt 3.1.2, kann eine Aktionsebene ihre Mutexe anhand ihrer Aktionen und der Propositionen aus der vorherigen Ebene berechnen und vice versa für Propositionsebenen. Auf Grund des Verfahrens der Graphenplanung, ist eine Ebene ebenfalls dazu in der Lage, die nächste Ebene selbst zu berechnen. Dieses Vorgehen wird in Programm 3.4 durch die Zeile

```
current = current.buildNextLayer()
```

erzielt. Eine Propositionsebene errechnet die nächste Aktionsebene anhand ihrer Propositionen und dem zuvor erstellten Aktionenbehälter. Dieser berechnet, welche Aktionen auf diese Propositionen angewendet werden können und erstellt damit die nachfolgende Ebene. Für eine Aktionsebene ist es ein Leichtes, die nächste Propositionsebene zu erstellen. Dafür müssen einfach die Auswirkungen der Aktionen dieser Ebene als Menge vereint werden, wobei Duplikate entfernt werden. Durch diese Struktur alternieren die Ebenen, wie im *GraphPlan* beschrieben, zwischen Propositionen und Aktionen.

Programm 3.4: Abstrahierte Implementierung des *GraphPlan* Algorithmus. Solange keine Lösung gefunden wurde, wiederholt sich die Routine und eine neue Ebene wird erstellt.

```
1 public Graphplan(ActionStore actionStore, Propositions start, Propositions goal) {
2   this.start = new PropositionLayer(start);
3   ...
4 }
5
6 public void solve() {
7   Layer current = this.start;
8
9   while(!this.solved) {
10    this.routine(current);
11
12    if(!this.solved) {
13      current = current.buildNextLayer();
14    }
15  }
16 }
```

In der Routine, welche in jedem Schritt ausgeführt wird, prüft der Algorithmus, ob bereits eine Lösung möglich wäre. Trifft der Algorithmus in dieser Routine auf eine Propositionsebene, welche alle nötigen Propositionen für den Endzustand beinhaltet, wird die Suche aktiviert. Dafür werden die Mutexe verwendet, welche beim Erstellen der Ebenen bereits berechnet wurden. Wird in diesem Schritt keine valide Lösung gefunden, wird die Ebene zu einer Liste hinzugefügt, die für die Abbruchbedingung verwendet wird, damit keine Endlosschleife zustande kommt. Wird eine Ebene zum zweiten Mal besucht, wird der Algorithmus abgebrochen und das Problem wurde ohne eine valide Lösung errechnet. Ansonsten wird die berechnete Lösung im Algorithmus hinterlegt, um im Weiteren verwendet werden zu können.

Kapitel 4

Von Graphen zur Lastenverteilung

Kapitel 3 ging konkret und detailliert auf den Mechanismus der Graphenplanung ein und auch, wie dieser allgemein auf verschiedene Szenarien angewendet werden kann. Interessant ist aber vor allem der Übergang eines Plans, der mit der Graphenplanung berechnet wurde, in ein verteiltes System zur Lastenverteilung.

4.1 Ausführungpläne

Bei der genaueren Betrachtung eines Planes, welcher durch den implementierten Algorithmus erstellt wurde, weißt dieser Aspekte der Lösungswege moderner Frameworks wie *Quartz* und *AKKA* auf. Dieser partiell geordnete Plan zeigt die Abhängigkeiten der einzelnen Aktionen auf und dadurch auch die Reihenfolge, in welcher diese ausgeführt wurden. In Abschnitt 3.3 wurde bereits eine Methode erläutert, die solche Aktionen in einem verteilten System als ausführbare Einheit, einem Task, implementierbar macht. Das bedeutet, das System kann berechnen, welcher Task von einem anderen abhängig ist und dessen Ergebnisse für weitere Berechnungen benötigt. Ein solcher Plan repräsentiert die Listener, die in *Quartz* manuell für jeden Job neu erstellt werden müssen, um ihn in kleinere Tasks aufzuspalten. Der Vorteil gegenüber den Listener ist der, dass die Logik für das Berechnen der Abhängigkeiten direkt im Task selbst deklariert wird. Dadurch bleiben die Zuständigkeiten klar strukturiert.

Zugleich können solche Pläne wie ein Baum betrachtet werden. Ausgehend vom Startzustand als Wurzel des Baumes, würde jede weitere Ebene die Tasks repräsentieren, welche parallel ausgeführt werden können. Zu beachten ist, dass Tasks einer Ebene nur dann ausgeführt werden können, wenn alle Tasks der darüber liegenden Ebene bereits beendet sind. Dieses Verhalten ähnelt der Hierarchie der Aktoren in *AKKA*. Wie in Abschnitt 2.1.2 beschrieben, können Aktoren in *AKKA* neue Aktoren erstellen, wie es ihnen beliebt, um die Arbeitslast weiter zu zergliedern. Im Endeffekt wird dadurch ein Baum erstellt, wobei Tasks durch Aktoren abgebildet werden und die Ausführungsreihenfolge nicht so strikt ist wie bei einem Plan. Das Aktorensystem ist dahingehend flexibler und kann aus diesem Grund auch in anderen Szenarien außerhalb der Lastenverteilung eingesetzt werden, wie zum Beispiel im *Play*[1] Framework, welches *AKKA* verwendet, um ein asynchrones Webframework zu implementieren.

[1] https://www.playframework.com/

4.2 Lastenverteilung anhand von Ausführungsplänen

Um einen Job, der durch seinen Ausführungsplan repräsentiert wird, in einem verteilten System auszuführen, wurde für den Prototypen eine Architektur gewählt, welche *Quartz* ähnelt. Die Entwickler bezeichnen *Quartz* oft als einen verteilten *CRON* bzw. einen Scheduler. Dieser ermöglicht das Erstellen eines sogenannten Schedules, um Jobs in gewissen Reihenfolgen bzw. zu gewissen Zeiten ausführen zu können. Für den Prototypen wurde ein solcher Scheduler entwickelt, der mithilfe des *GraphPlan* Algorithmus einen Ausführungsplan für ein Problem berechnet und die einzelnen Tasks, aus denen er besteht, auf einer oder mehrerer Berechnungseinheiten ausführen lässt. Im einfachsten Fall eines lokalen Schedulers wäre der Scheduler lediglich ein Thread, der Tasks mithilfe eines Threadpools berechnet. Dabei werden die Ergebnisse von einem Task zum nächsten weitergegeben, sofern diese voneinander abhängig sind und zeitgleich auch ihre Ausführungsreihenfolge gewährleistet.

Abstrakt gesehen, kann dieses Verfahren mit dem Scheduler eines Betriebssystems verglichen werden. Dieser bietet viele weitere Funktionalitäten, jedoch gibt es Szenarien, in denen sich beide sehr ähnlich sind. Wird zum Beispiel in Linux in einem Programm ein IO Befehl ausgeführt, wie das Lesen einer Datei von der Festplatte, geht der Thread, in Linux als Prozess bezeichnet, in dem dieses Programm ausgeführt wird, in einen Schlafmodus [23, S. 141]. Diesen verlässt er solange nicht, bis die Ressourcen, die er benötigt, frei sind. Mit der Graphenplanung kann diese Herangehensweise mit zwei Aktionen verglichen werden, wobei die erste `DateiLesen(f)` die Datei ausliest und die zweite `BearbeiteDaten(d)` die Daten der Datei bearbeitet. Der Task `BearbeiteDaten` bleibt solange im Schlafmodus, bis der Scheduler die Daten der Datei vom Task `DateiLesen` zur Verfügung hat, um im Weiteren diese verarbeiten zu können. Wie ein solcher Scheduler, anstatt wie im einfachsten Fall lokal, in einem verteilten System arbeitet, wird im nächsten Kapitel beschrieben.

Kapitel 5

Systemarchitektur

Dieses Kapitel widmet sich der grundlegenden Architektur des verteilten Systems, welches für den Prototypen entwickelt wurde. Dabei werden im ersten Schritt die einzelnen Komponenten mit ihren Zuständigkeiten und Verantwortungen, aufgezeigt und erläutert. Ein wichtiger Aspekt dieser Architektur, welcher auch behandelt wird, ist die Kommunikation und welche Protokolle dafür in Anwendung sind bzw. wie diese angepasst wurden.

5.1 Komponenten der Architektur

Im Grunde kann die Systemarchitektur in drei Komponenten aufgeteilt werden, wobei jede dieser Komponenten genau ein spezielles Teilgebiet des Netzwerkes beschreibt. Die Komponenten und ihr Zusammenspiel wurde für den Prototypen recht einfach gehalten, damit die Implementierung als Diskussionsgrundlage dieser Arbeit dienen kann.

5.1.1 Registry

Die erste Komponente des Netzwerkes ist die sogenannte Registry. Ins Deutsche übersetzt bedeutet Registry eine Eintragung bzw. Sekretariat, was ihrer Funktion sehr nahe kommt. In einer Registry können Einträge vorgenommen werden, die gewisse Metadaten beinhalten. Im Prinzip ist sie nichts anderes als ein Programm, welches in einem Netzwerk über ihre IP-Adresse und ihren Port ansteuerbar ist und in die Daten geschrieben werden können. Diese Daten können im Weiteren von anderen Komponenten des Netzwerkes ausgelesen werden. Dabei handelt es sich aber nicht um Daten, die für die Verarbeitung gedacht sind, sondern ähneln sehr stark einem Quellenverzeichnis. Denn die Einträge verweisen auf eine andere Komponente im Netzwerk und beinhalten Informationen darüber, wie diese angesprochen werden kann. Im Grunde registrieren sich Komponenten in einem verteilten System bei einer Registry, damit sie von anderen Komponenten gefunden werden können. In der Softwareentwicklung spricht man oft vom sogenannten *Service-Locator Pattern* [28], denn die Komponente wird dabei mit einer Schnittstellendefinition abstrahiert, um einfacher ausgetauscht werden zu können. Der Vorteil liegt darin, dass Komponenten einen zentralisierten Weg haben, andere im Netzwerk zu finden. Um die Funktionalität des Systems zu gewährleisten, sollten Registries immer geklont werden, damit bei Ausfällen reagiert werden kann.

Programm 5.1: Beispiel einer Problembeschreibung und wie diese an den Scheduler übermittelt werden kann.

```
1  SchedulerInterface scheduler = new SchedulerInterface(ip, port);
2
3  TaskState state = new TaskState();
4
5  state
6    .has(new UserId(999999))
7    .wants(Friends.class);
8
9  scheduler.schedule(state);
```

5.1.2 Scheduler

Die zentrale Rolle des Netzwerkes spielt der Scheduler, dieser bildet die zweite Komponente. Der Scheduler registriert sich bei der Registry, um von anderen Komponenten gefunden werden zu können. Außerdem bildet er die Schnittstelle für externe Applikationen, welche das Lastenverteilungssystem nutzen wollen. Es gibt zur Laufzeit immer nur einen Scheduler, der sich bei der Registry einträgt. Wie ein derartiges System auf Ausfälle des Schedulers reagiert, um die Ausfallzeit zu minimieren und die Funktion aufrecht zu erhalten, wird in Abschnitt 7.2.4 erklärt. Um einen Job im System abarbeiten zu lassen, muss eine externe Applikation bei der Registry nach der Schnittstelle des Schedulers nachfragen. Mit dieser kann die Applikation Problembeschreibungen an den Scheduler übermitteln. Eine Problembeschreibung besteht aus dem Startzustand und dem gewünschten Zielzustand. Programm 5.1 zeigt auf, wie eine derartige Problembeschreibung erstellt werden kann, am Beispiel von *SYNCED*, zum Abfragen der Freunde von den externen Diensten wie *Facebook*, *Google+* und anderen. Wird eine solche Problembeschreibung an den Scheduler übermittelt, kann dieser anhand der implementierten Tasks einen Plan erstellen. Die einzelnen Tasks des Plans werden in den Schedule aufgenommen und vorerst auf den Status *HOLD* gesetzt. Im nächsten Schritt werden alle Tasks, die noch im Status *HOLD* sind und bereits ausgeführt werden können, auf die dritte Komponente des System delegiert, die Abarbeitungseinheiten (Worker). Tasks können genau dann ausgeführt werden, wenn alle ihre Voraussetzungen zutreffen und alle Abhängigkeiten aufgelöst wurden.

Wurde ein Task abgearbeitet, ändert der Scheduler den Status des Tasks auf *FINISHED*, sofern keine Fehler aufgetreten sind. Die Ergebnisse werden in den derzeitigen Zustand des gesamten Jobs übertragen, damit weitere Tasks auf diese Zugriff haben. Tritt während der Abarbeitung ein Fehler auf, wird der Status des Tasks auf *FAILED* gesetzt, mit der zusätzlichen Information des Fehlers, der aufgetreten ist. Wie diese Fehlerinformationen im Weiteren verwendet wird, wird in Kapitel 7 erläutert. Wenn alle Tasks eines Jobs abgearbeitet wurden, werden die Ergebnisse, welche für den Zielzustand benötigt werden, gesammelt und an die Applikation zurückgeschickt, die den Job beauftragt hat.

5.1.3 Worker

Die letzte Komponente der Architektur bilden die Abarbeitungseinheiten. Diese verbinden sich beim Start, anhand der Registry, mit dem Scheduler und registrieren sich dort, um im Weiteren Tasks übermittelt zu bekommen. Im Gegensatz zur Registry ist der Kommunikationskanal hier bidirektional, um Ergebnisse an den Scheduler zurück zu übermitteln. Somit bleibt der Kommunikationskanal dauerhaft bestehen. Worker besitzen intern einen Threadpool, auf dem sie Tasks parallel ausführen können. Erhält ein Worker einen Task vom Scheduler, versucht er diesen sofort auszuführen, was jedoch nicht immer sofort möglich ist. Sind zum Beispiel keine Ressourcen in Form von Threads verfügbar, setzt der Worker den Task auf eine Warteschlange und informiert den Scheduler darüber, was für die Lastenverteilung wichtig ist. Werden Ressourcen wieder frei, wird der Task auf einem Thread ausgeführt. Wie zuvor erwähnt, stellt ein Task zwei Methoden zur Verfügung, mit denen signalisiert werden kann, ob die Ausführung fehlerfrei war oder nicht, welche an dieser Stelle zum Einsatz kommen, um dem Scheduler Auskunft darüber zu geben. Dieser kann im Weiteren darauf reagieren, um den Task abzuschließen bzw. auf Fehlerfälle einzugehen. Damit der Thread, in dem der Task selbst ausgeführt wird, keine Kommunikation mit dem Scheduler aufbauen muss, legen die beiden Methoden einen Event auf eine Warteschlange, die dann im Hauptthread abgearbeitet wird. Nähere Informationen zu dieser Funktionsweise gibt es im nächsten Abschnitt.

5.2 Kommunikationsparadigmen

Viele verteilte System basieren auf dem Austausch von Nachrichten zwischen den einzelnen Komponenten. Dadurch wird jedoch die Kommunikation nicht abstrahiert, was zu komplexeren Fehlerbehandlungen führt. Um den Nachrichtenaustausch zu abstrahieren und zu vereinfachen, kommt im Prototypen das Prinzip des entfernten Prozeduraufrufes (*Remote Procedure Call*, RPC) zum Einsatz.

5.2.1 Entfernter Prozeduraufruf

Der Grundgedanke von RPC ist es, dass Programme die Möglichkeit haben, Prozeduren auf anderen Rechnern aufzurufen. Wenn zum Beispiel das Programm eines Rechners A eine Prozedur auf Rechner B per RPC ausführt, wird das Programm auf Rechner A angehalten und auf dem Rechner B weitergeführt. Dabei erscheint es im Programm wie ein normaler Aufruf einer lokalen Prozedur. Dieses Verhalten wird bei RPC durch sogenannte Stubs realisiert. Ein Stub ist nichts anderes als eine Prozedur, die wie eine Referenz auf die entfernte Prozedur funktioniert. Dabei verpackt sie übergebene Parameter in eine Nachricht und sendet sie an den entfernten Rechner, um dort die eigentliche Prozedur auszuführen [24, S. 150].

5.2.2 Entfernter Methodenaufruf

In Java gibt es RPC in diesem Sinne nicht. Javas Gegenstück zu RPC ist der objektorientierte entfernte Methodenaufruf (*Remote Method Invocation*, RMI). Dabei bilden Stubs nicht eine Referenz einer Prozedur, sondern eines entfernten Objektes. Dieses

Objekt wird auf dem entfernten Rechner erstellt und implementiert dabei eine gewisse Schnittstelle. Anschließend kann es in einer Registry abgelegt werden und von anderen Komponenten ausgelesen werden. Auf diesen Stub können dann die entfernten Methoden der Schnittstelle ausgeführt werden, als wäre es ein lokales Objekt. Im Fall des Prototypen wird die Registry nicht direkt auf dem Rechner angelegt, auf dem der Scheduler ausgeführt wird, sondern wie zuvor erwähnt direkt in die Registry Komponente gelegt. Diese ist danach imstande den Scheduler Stub an andere Komponenten auszuliefern. Außerdem hat sie im Netzwerk dadurch eine zweite wichtige Rolle, die in Abschnitt 7.2.4 genauer erläutert wird, um die Ausfallsicherheit des Systems aufrecht zu erhalten.

5.2.3 Eventloop statt Multithreading

Zu berücksichtigen ist, dass jeder Zugriff, der über RMI in Java durchgeführt wird, in einem neuen Thread abgearbeitet wird. Das klingt im ersten Moment sehr trivial, eröffnet aber im Weiteren Probleme. Wenn zum Beispiel eine externe Applikation eine Problembeschreibung an den Scheduler mittels RMI übermittelt, wird dadurch am Scheduler ein Thread geöffnet. In diesem Thread wird dann der Plan berechnet und die einzelnen Tasks in die Datenbank geschrieben. Danach werden die Tasks an die Worker übermittelt und abgearbeitet. Wenn der letzte Task fertig ist, ist die entfernte Methode abgeschlossen und das Ergebnis könnte via RMI an die Applikation übergeben werden. Das bedeutet aber, dass der Aufruf solange dauert, wie das verteilte Abarbeiten des Plans. Außerdem wäre für jeden Aufruf ein offener Thread am Scheduler, der Ressourcen benötigt und sich mit anderen Threads Ressourcen teilt, die synchronisiert werden müssten.

Viele moderne Applikationen wie Node.js entfernen sich immer weiter von diesem Modell und versuchen mit weniger bzw. nur einem Thread dieselbe Leistung zu erzielen. Diese Herangehensweise macht sie in der Breite viel skalierbarer, da einfach eine weitere Instanz am selben Server hinzu geschaltet werden kann, um die Leistung zu erhöhen [25]. Der Prototyp baut sehr stark auf diesem Prinzip auf. Die einzelnen Komponenten der Architektur wurden im Prototypen nach dem Eventloop Pattern entwickelt. Das bedeutet, dass jede Komponente mit einem einzelnen Thread realisiert wurde. Dieser durchläuft, ähnlich dem Hauptloop eines Spieles, ständig eine Routine, in dem alle Ereignisse (Events), die seit dem letzten Durchlauf eingetreten sind, abgearbeitet werden. Typischerweise hat ein Thread, der auf einem Eventloop basiert, eine Warteschlange, auf die Events gelegt werden können. Events sind gekennzeichnet durch einen Schlüssel bzw. Namen und lösen direkt im Eventthread eine Callback-Methode aus, um auf das Event zu reagieren. Ein derartiges System neutralisiert mit seiner Funktionsweise bereits alle Wettlaufsituationen, die in parallelen Systemen auftreten können, da nie mehr als ein Thread auf dieselbe Ressource gleichzeitig zugreifen kann, abgesehen von der Warteschlange des Eventthreads.

Übermittelt zum Beispiel eine externe Applikation wiederum eine Problembeschreibung an den Scheduler, wird im erstellten Thread des RMI-Aufrufs nur der Plan berechnet, damit bei keiner vorhandenen Lösung direkt geantwortet werden kann. Der Plan wird dann in ein Eventobjekt verpackt und auf die Warteschlange des Eventthreads gelegt. Damit wäre die Methode bereits fertig und die externe Applikation bekommt eine Promise zurück. Diese wird aufgelöst, sobald alle dazugehörigen Tasks fertig sind

bzw. zurückgewiesen im Falle eines Fehlers. Die restliche Logik, wie zum Beispiel das Persistieren der Tasks oder das Verteilen an die Worker regelt der Eventthread, auf dem der Eventloop abgearbeitet wird. Damit Netzwerkzugriffe auf andere Komponenten im System bzw. Datenbankzugriffe den Eventloop nicht blockieren, kann auf *RxJava* zurückgegriffen werden. *RxJava* ist die Implementierung der reaktiven Erweitung[1] für Java, womit viele Funktionalitäten wie Datenbankzugriffe und RMI asynchron behandelt werden können. Ein weiterer Vorteil dieses Modells ist die Möglichkeit der externen Applikation währenddessen weiterarbeiten zu können und benachrichtigt zu werden, sobald die Abarbeitung fertig ist, anstatt wie beim herkömmlichen RMI zu blockieren.

5.2.4 Asynchroner entfernter Methodenaufruf

Die Implementierung von RMI in Java ist von Grund auf synchron und blockiert die Applikation. Wie in JavaScript gibt es aber auch in Java die Möglichkeit, eine Promise, in Java als Future bezeichnet, zu definieren, um auf ein Ergebnis zu reagieren, wenn es verfügbar ist, anstatt blockierend darauf zu warten. Programm 5.2 zeigt den Unterschied zwischen einer blockierenden und nicht blockierenden I/O-Operation. Der Nachteil von RMI gegenüber einer herkömmlichen Websocket-Kommunikation ist der, dass bei RMI nur in eine Richtung kommuniziert wird. Wenn zum Beispiel eine externe Applikation eine RMI-Verbindung mit dem Scheduler aufbaut, kann die Applikation Methoden auf den Scheduler aufrufen, jedoch der Scheduler nicht auf die Applikation. Es ist aber möglich anhand der Information aus der Verbindung auf die selbe Art und Weise eine RMI-Verbindung zurück zur Applikation aufzubauen. Jedoch muss gewährleistet sein, dass die Applikation einen dafür geöffneten Port zur Verfügung stellt. Oftmals agieren solche Applikationen aber auf privaten Rechnern, die per IP und Port nicht ansteuerbar sind bzw. eine Firewall haben, die die Verbindung blockiert. Um solche Situationen der Standardimplementierung von Javas RMI zu lösen, kann in der Applikation ein Eventthread gestartet werden, der die Verbindung zum Scheduler verwendet, um eine weitere spezielle Methode am Scheduler aufzurufen. Der Thread, welcher am Scheduler für diesen Aufruf erstell wird, wird direkt in den Schlafmodus versetzt. Wurde ein Job, der von dieser Applikation angefordert wurde, erfolgreich ausgeführt bzw. kam es zu einem Fehler, kann der Scheduler den dazugehörigen Thread aufwecken, um die Future auf der Applikationsseite aufzulösen bzw. zurückzuweisen. Wie zuvor erwähnt, gibt es mittlerweile per *RxJava* die Möglichkeit, RMI asynchron aufzurufen. Dabei überschreibt *RxJava* die Standardimplementierung von Java um dies zu ermöglichen, bietet dadurch aber eine weit aus sauberere Lösung.

[1] http://reactivex.io

Programm 5.2: Unterschied zwischen synchronen und asynchronen I/O-Operationen.

```
1  // Aufruf blockiert das Programm, bis die I/O-Operation fertig ist.
2  String content = readFileSynchronously(filename);
3  System.out.print(content);
4
5  // Aufruf blockiert nicht, führt die Callback-Methode aus, die bei
6  // then(...) übergeben wird, sobald die Datei fertig gelesen wurde.
7  CompleteableFuture<String> future = readFileAsynchronously(filename);
8  future.thenAccept(System.out::print);
9
10 // Code der hier folgt kann ausgeführt werden, während
11 // die Datei ausgelesen wird.
12 System.out.print("Wahrscheinlich werde ich vor dem Inhalt der Datei geschrieben");
```

Kapitel 6

Lastenverteilung

Die Lastenverteilung ist der Prozess in einem verteilten System, bei dem die Lasten auf den Knoten verteilt bzw. neu verteilt werden, um die Leistung des Systems zu erhöhen. Die primären Ziele der Lastenverteilung sind die Ressourcen optimal zu verwenden, den Durchsatz zu maximieren (Anzahl der ausgeführten Tasks), die Antwortzeiten zu minimieren und zu verhindern, dass ein Knoten im System überladen wird, ohne zusätzliche Hardware (Knoten) einsetzen zu müssen [11, 15]. Die Knoten in einem verteilten System können oft unterschiedliche Rechenkapazitäten besitzen, vor allem in Cloud- und Gridumgebungen. Das bedeutet, dass die Arbeitslast unter Berücksichtigung der einzelnen Rechenkapazitäten gleichmäßig verteilt werden muss, damit Leerläufe auf Knoten verhindert werden und alle Tasks in minimaler Zeit abgearbeitet werden können. In der Lastenverteilung gibt es einige Themen, die berücksichtigt werden müssen [15]:

1. Die Kommunikationskanäle zwischen Knoten in einem verteilten System sind in ihrer Zahl und Kapazität begrenzt. Außerdem können die einzelnen Knoten geographisch weit verbreitet sein. Diese Punkte müssen bei der Lastenverteilung einfließen, wenn entschieden werden soll, einen Task von einem Knoten zu einem anderen zu migrieren.
2. Ein Job kann nicht in beliebig viele Tasks geteilt werden, was zu gewissen Beschränkungen in der Verteilung führt.
3. Jeder Job besteht aus kleineren Tasks, wobei diese voneinander abhängig und unterschiedliche Ausführungszeiten haben können.
4. Die Last auf den einzelnen Knoten, sowie ihres Netzwerkes, können über Zeit variieren, je nach Anzahl der Anfragen.
5. Die Rechenkapazität der einzelnen Knoten kann durch die Architektur, Betriebssystem, CPU und Speicher variieren.

Ausgehend von diesen Punkten besteht die Aufgabe der Lastenverteilung darin, die einzelnen Knoten zu überwachen und Lasten gleichmäßig zu verteilen, sowie Lasten zwischen Knoten zu migrieren. Zudem müssen auch Daten, wie zum Beispiel Dateien, die für die Ausführung der Tasks wichtig sind, beim Migrieren ebenfalls mitübertragen werden. Aufgrund der großen Vorteile der Lastenverteilung, wie Antwortzeiten minimieren und das Optimieren der Ressourcen, wurde in diesem Gebiet sehr aktiv geforscht [15]. In diesem Kapitel werden im Weiteren Ansätze der Lastenverteilung erläutert.

6.1 Statische und Dynamische Lastenverteilung

In der Regel kann die Lastenverteilung in zwei Ansätze kategorisiert werden, die statische und die dynamische Lastenverteilung. In der statischen Lastenverteilung werden die Entscheidungen für die Verteilung bereits bei der Kompilierung getroffen. Das bedeutet, dass diese oft basierend auf Wahrscheinlichkeiten bzw. deterministisch getroffen werden und im Vorfeld definiert werden muss, wie viel Leistung die einzelnen Knoten erbringen können. Die statische Lastenverteilung baut auf der Annahme, dass die Leistung der Knoten während der Laufzeit nahezu konstant bleibt. Dadurch werden Lasten nicht migriert, weil sie laut der Theorie bereits gleichmäßig verteilt sind. Im Gegensatz dazu arbeitet die dynamische Lastenverteilung zur Laufzeit. Dabei wird ständig das System überwacht und Lasten neu berechnet sowie zwischen den Knoten migriert. Dabei kann die dynamische Lastenverteilung in verschieden Kategorien eingeteilt werden [15], wobei in dieser Arbeit nur auf die zentralisierten Algorithmen eingegangen wird, weil diese im Prototypen eingesetzt werden können.

Im zentralisierten Ansatz der Lastenverteilung gibt es einen zentralen Knoten (Scheduler). Dieser hat Informationen über alle Tasks, die ausgeführt werden sollen bzw. bereits ausgeführt werden. Außerdem hat er einen Überblick über das gesamte System und entscheidet, anhand von Informationen über die Auslastung der einzelnen Knoten, wie die Tasks verteilt werden sollen. Die Tasks werden dann auf die Worker zur Ausführung übertragen. Der große Vorteil ist der simple Aufbau des Systems, jedoch leiden derartige Ansätze an dem Problem des einzelnen Bruchpunktes (*single point of failure*). Im Prototyp wurde ein solcher Algorithmus entwickelt, der in den Abschnitten 6.2.3 und 6.3.1 genauer erläutert wird.

6.2 Algorithmen der statischen Lastenverteilung

Algorithmen der statischen Lastenverteilung überprüfen zur Laufzeit nicht die Auslastung der einzelnen Knoten, sondern verteilen die Last deterministisch. Unter anderen zählen die folgenden Algorithmen zur statischen Lastenverteilung.

6.2.1 Round Robin Algorithmus

Der Round Robin Algorithmus verteilt die Lasten sequentiell und gleichmäßig auf die Knoten [15]. Dabei werden die Knoten als eine zyklische Liste gesehen, die wieder von vorne beginnt, wenn das Ende erreicht wird. Das bedeutet, dass der erste Task dem ersten Knoten in dieser Liste zugeteilt wird, der zweite Task dem zweiten Knoten usw. Der große Vorteil ist die nicht benötigte Kommunikation zwischen dem Scheduler und den Workern, um die derzeite Auslastung zu berechnen. Round Robin funktioniert jedoch nur effektiv in Umgebungen, in denen alle Worker dieselbe Leistung erzielen können und alle Tasks beinahe gleiche Ausführungszeiten aufweisen.

6.2.2 Randomized Algorithmus

Beim Randomized Algorithmus werden zufällige Zahlen verwendet, um zu entscheiden, auf welchen Knoten die nächste Last verteilt wird. Dabei wird aber außer Acht gelassen, wie der Knoten derzeitig ausgelastet ist. Den einzelnen Knoten werden Zahlen

zugewiesen und die Zufallszahl wird anhand von statistischer Verteilung [15] ermittelt. Der Randomized Algorithmus ist nur in speziellen Szenarien wirklich effektiv, wird aber oft als eine nicht elegante Lösung abgestempelt. Außerdem weist er unter den statischen Algorithmen die höchste Antwortzeit auf.

6.2.3 Central Manager Algorithmus

Der Central Manager Algorithmus benötigt, wie der Name bereits beschreibt, einen zentralen Koordinator. Der Koordinator verteilt in jedem Schritt einen Task an genau einen Worker im System. Dabei wählt er den Worker mit der derzeit niedrigsten Anzahl an Tasks. Das bedeutet, der Koordinator behält eine Übersicht über die einzelnen Tasks und auf welchen Workern diese ausgeführt werden, so wie der Scheduler im Prototypen. Sofern sich die Last verändert, zum Beispiel ein Task fertig bearbeitet wurde, sendet der Worker dem Koordinator eine Nachricht. Anhand dieses Verhaltens, trifft der Koordinator immer die bestmöglichen Entscheidungen, sofern die Tasks ähnliche Ausführungszeiten aufweisen. Der Nachteil liegt wie in Abschnitt 6.1 bereits beschrieben im *single point of failure*, den der Koordinator darstellt.

6.3 Algorithmen der dynamischen Lastenverteilung

Im Gegensatz zur statischen Lastenverteilung, fließen bei der Dynamischen zusätzliche Laufzeitinformationen in die Entscheidungen ein, auf welchen Knoten welche Tasks ausgeführt werden sollen. Diese Algorithmen adressieren Probleme, welche mit der statischen Lastenverteilung nicht möglich sind, zum Beispiel Umgebungen, in denen Tasks unterschiedliche Ausführungszeiten aufweisen bzw. sich über die Laufzeit ändern können. Algorithmen der dynamischen Lastenverteilung überwachen und dokumentieren Änderungen im System und reagieren angemessen darauf, indem Tasks neu verteilt und migriert werden können. Sie sind, verglichen mit den statischen Algorithmen, komplizierter und weisen einen höheren Kommunikationsoverhead auf.

6.3.1 Central Queue Algorithmus

Beim Central Queue Algorithmus werden neue Tasks und Anfragen in einer zyklischen Warteschlange (Queue) gespeichert, die auf einem zentralen Koordinator, wie beim Central Manager Algorithmus, liegt. Wenn ein Worker unter einen Auslastungs-Threshold fällt und wieder Ressourcen zur Ausführung übrig sind, wird beim Koordinator nach einem neuen Task nachgefragt. Wenn in der Warteschlange ein Task bereit liegt, wird dieser an den Worker geschickt, ansonsten wird die Anfrage des Workers ebenfalls in der Warteschlange abgelegt. Wird ein neuer Task in die Warteschlange gelegt und es existiert bereits eine Anfrage eines Workers, wird dieser Task sofort zu diesem Worker geschickt. Die Kommunikation ist höher als beim Central Manager Algorithmus, jedoch wird Rücksicht darauf genommen, ob ein Worker noch über Ressourcen zur Ausführung verfügt oder nicht.

6.3.2 Local Queue Algorithmus

Der Local Queue Algorithmus unterstützt das Prinzip der Taskmigration. Bei diesem Algorithmus gibt es keinen zentralen Koordinator, sondern nur Worker. Wenn ein Job ausgeführt werden soll, wird er an einen Worker geschickt, der noch nicht komplett ausgelastet ist. Jeder dieser Worker hat einen lokalen Lastenmanager, der erkennen kann, ob ein Worker ausreichend ausgelastet ist oder nicht. Die einzelnen Tasks eines Jobs werden in diesen lokalen Manager gelegt. Wenn der Manager erkennt, dass der Worker unter eine gewisse Anzahl von ausführbaren Tasks fällt (Threshold), sendet er eine Anfrage an andere Worker. Wenn ein Worker eine solche Anfrage erhält, vergleicht er, wieviele Tasks er noch übrig hat mit der Anzahl der Tasks, die der unter-belastete Worker angefragt hat. Sollte er mehr Tasks übrig haben, als angefragt wurden, werden diese Tasks an den unter-belasteten Worker übermittelt und migriert.

6.3.3 Honey Bee Foraging Algorithmus

Dieser Algorithmus ist besonders interessant, da er von der Natur inspiriert wurde. Dabei handelt es sich um einen selbst organisierenden Algorithmus, welcher häufig in dynamischen Cloud Umgebungen verwendet wird [11]. Der Algorithmus baut auf dem Verfahren auf, welches Honigbienen bei der Futtersuche verwenden. Dabei werden Arbeiterbienen zur Futtersuche ausgesendet, um Futterquellen zu finden. Wenn eine neue Quelle gefunden wurde, kehrt die Biene zurück in den Stock und präsentiert ihren Fund mit einem Tanz. Die Qualität der Quelle wird dabei definiert durch die Quantität und Qualität des gefundenen Nektars und auch, wie weit die Quelle vom Stock entfernt ist. Die gesamte Information dafür steckt im Tanz der Arbeiterbiene. Danach folgen mehrere Arbeiterbienen der Biene, welche die Quelle gefunden hat, und sammeln soviel Nektar wie möglich. Nachdem diese Bienen wieder im Stock ankommen, teilen sie den anderen Bienen wiederum mit einem Tanz mit, wie viel Nektar noch übrig ist, der gesammelt werden kann, wodurch weitere Bienen ausschwärmen, bis die Quelle komplett ausgeschöpft wurde. Dieses Verfahren wird immer häufiger auf Lastenverteilungssystem angewendet [21]. Dabei wird ein *Profit* errechnet, wenn ein Task ausgeführt worden ist, wie die Qualität einer gefundenen Quelle, anhand der Kosten, die ein Worker dafür aufwenden musste, wie zum Beispiel die CPU-Zeit oder Speicheraufwand. Der Tanz der Bienen wird mittels eines verteilten Speichers umgesetzt, um mit den anderen Knoten (Bienen) zu kommunizieren. In diesem Speicher wird auch der globale *Profit* berechnet, um zu beobachten wie effektiv Ressourcen verwendet werden. Wie die Architektur eines solchen Systems aufgebaut sein muss, würde den Rahmen dieser Arbeit sprengen, wird aber in [21] genauer beschrieben.

6.4 Lastenverteilung im Prototypen

Hinsichtlich des Prototypen wurden Aspekte der statischen und dynamischen Lastenverteilung kombiniert. Wie zuvor in Abschnitt 6.1 erwähnt, wurden für den Prototypen nur zentralisierte Algorithmen herangezogen, auf Grund der zentralisierten Architektur des Systems. Aus Kapitel 5 geht hervor, dass der grundlegende Algorithmus der Lastenverteilung auf dem Central Manager-Algorithmus aus Abschnitt 6.2.3 aufbaut. Der Scheduler bildet dabei den Koordinator und verteilt die einzelnen Tasks gleichmäßig an

die registrierten Worker. Jedoch können sich die einzelnen Tasks in ihrer Ausführungszeit sehr stark unterscheiden. Ein Aufruf eines externen Dienstes braucht zum Beispiel viel länger als das Zusammenführen der einzelnen Freundeslisten. Das bedeutet aber auch, dass ein rein statischer Algorithmus für die Lastenverteilung nicht ausreichend ist. Aus diesem Grund wurde der Algorithmus etwas abgeändert.

Jeder Worker verfügt lokal über eine Warteschlange. Erhält ein Worker einen Task, für den er keine Ressource zur Ausführung übrig hat, wird dieser auf die Warteschlange gelegt. Außerdem wird der Scheduler über diesen Zustand in Kenntnis gesetzt. Der Scheduler überwacht die einzelnen Warteschlangen der Worker und verteilt neue Tasks an den Worker mit den wenigsten übrigen Tasks. Sollten alle Warteschlangen der Worker einen definierten Threshold übertreten, müsste ein weiterer Worker im System registriert werden, um die Lasten abarbeiten zu können. Der Vorteil der Warteschlangen liegt darin, dass diese Tasks einfach auf neu registrierte Worker migriert werden können. Eine weitere Möglichkeit, den Algorithmus zu beschleunigen, wäre die einzelnen Warteschlangen, ähnlich dem Central Queue Algorithmus aus Abschnitt 6.3.1, direkt am Scheduler zu implementieren, damit der Überwachungs- und Migrierungsmechanismus einfacher und zentralisierter abgehandelt werden kann. Das bedeutet aber auch, dass Ausfälle des Schedulers sehr gravierend für das System wären. Wie derartige Ausfälle maskiert werden können, ohne Informationen zu verlieren und Leistung einzubüßen, wird in Abschnitt 7.2.4 genauer erläutert.

Kapitel 7

Fehlertoleranz

Fehlertoleranz ist ein grundlegendes und wichtiges Thema eines verteilten Systems. Dabei wird in dieser Arbeit auf zwei konkrete Themen der Fehlertoleranz eingegangen. Im ersten Abschnitt wird auf das Behandeln von Fehlern bei der Ausführung von Tasks eingegangen. Im Weiteren wird die eigentliche Fehlertoleranz des verteilten Systems erläutert.

7.1 Fehlermodelle in der Ausführung

Wie zuvor erwähnt, kann es bei der Ausführung eines Tasks immer wieder zu Fehlern kommen, weshalb es auch so wichtig ist, im ersten Schritt Jobs in einzelne Tasks aufzulösen, um die Fehlerbehandlung zu vereinfachen. Tritt im Prototypen ein Fehler während der Ausführung auf, übermittelt der Worker diesen Fehler an den Scheduler. Bevor ein Job an den Scheduler gesendet wird, kann definiert werden, wie im Falle eines Fehlers darauf reagiert werden soll. Der einfachste Weg, um auf einen Fehler zu reagieren, ist es, die weitere Ausführung des Jobs abzubrechen. Hierbei wird jedoch, bis auf die feiner granulierte Parallelität, kein weiterer Nutzen aus den kleineren Tasks gezogen, verglichen zu einem großen Job. Der große Vorteil der Tasks kommt zur Geltung, wenn auf Fehler konkret eingegangen werden kann bzw. soll. Wird zum Beispiel in einem Task eine Anfrage mit der *Google* Java Bibliothek ausgeführt, um Daten eines Benutzers von *Google+* oder *Youtube* abzurufen, kann es oft vorkommen, dass die benötigten Anmeldedaten abgelaufen sind. Um trotzdem an die Daten zu kommen, müssen vorher die Anmeldedaten aktualisiert und ein neuer Zugriffstoken erstellt werden. Danach funktioniert der Aufruf und die Daten können erfolgreich abgefragt werden. Um dieses Verhalten zu erzielen, gibt es die folgenden Fehlermodelle in Bezug auf die Ausführung.

7.1.1 Erneutes Versuchen

Eine typische Herangehensweise an einen Fehler, auch außerhalb von verteilten Systemen, ist es, erneut zu versuchen, was zuvor nicht funktioniert hat. Möglicherweise kam es nur zu einer vorübergehenden Störung (transient fault) und ein erneutes Versuchen würde zum gewünschten Ergebnisses führen. Wie im zuvor aufgezeigten Beispiel, kann es aber auch wiederkehrende Fehler geben, die zuvor behandelt werden müssen, bevor ein erfolgreiches Ausführen gewährleistet werden kann. Dafür bietet ein Task zusätzlich

Programm 7.1: Das Überschreiben der `rescue` Methode bietet die Möglichkeit auf Fehler einzugehen und zu beheben, bevor erneut versucht wird den Task auszuführen.

```
1  class FetchDataFromGoogle extends Task {
2    @Override
3    public void rescue() {
4      if(this.getReason() instanceof InvalidTokenException) {
5        // neuen Token bei Google anfragen
6      }
7    }
8  }
9
10 // externer Applikation muss die Strategie angeben, im Falle eines Fehlers
11 TaskState state = new TaskState();
12
13 state
14   .has(new UserId(999999))
15   .wants(Friends.class)
16   .with(Strategy.Rerun);
```

die Möglichkeit, eine gewisse Methode zu überschreiben, welche ausgeführt wird, wenn es zuvor zu Fehlern gekommen ist. Diese Methode kann im Weiteren prüfen, ob sie in der Lage ist, den aufgetretenen Fehler zu bereinigen, zum Beispiel durch das Erneuern des Zugriffstokens der *Google* Bibliothek. Dadurch kann sich der Task vor dem erneuten Versuchen selbst erholen und ermöglicht ein erfolgreiches Ausführen. Kann der Task jedoch nicht auf den Fehler reagieren, kann definiert werden, wie oft versucht werden soll, ihn erneut auszuführen, bis der gesamte Job abgebrochen wird. Programm 7.1 zeigt, wie ein Task auf Fehler reagieren kann, und wie bei einer Problembeschreibung die Strategie gesetzt werden kann.

7.1.2 Neustart

Es kann auch vorkommen, dass ein Job mit zeitkritischen Daten arbeitet, zum Beispiel beim Auswerten von Statistiken oder Ähnlichem. Das bedeutet, dass das erneute Versuchen von einzelnen Tasks die Daten verfälschen könnte. Um auf solche Situationen einzugehen, gibt es die Möglichkeit, die Strategie zu wählen, welche einen gesamten Job von vorne beginnt. Dabei können vereinzelte Tasks natürlich wieder auf Fehler eingehen, um ein erfolgreiches Ausführen zu gewährleisten. Der große Unterschied ist jedoch, dass ein gesamter Job und damit auch alle seine Tasks zurückgesetzt werden, sobald ein Fehler in einem Task auftritt. Programm 7.2 zeigt, wie diese Strategie bei einer Problembeschreibung angewendet werden kann.

7.1.3 Zurückrollen

Die letzte Strategie ist ähnlich wie ein Zurückrollen (Rollback) bei Transaktionen. Diese Strategie ist wichtig, wenn gewisse Tasks bereits Änderungen in Datenbanken oder anderen Dateien vornehmen, die bei einem Fehler wieder zurückgesetzt werden müssen. Es gibt auch Möglichkeiten, um solche Szenarien gänzlich zu umgehen, indem ein

Programm 7.2: Die Strategie `RESTART` führt zum erneuten Ausführen des gesamten Jobs, im Falle eines Fehlers.

```
1  state
2    .has(new StatisticArgs(...))
3    .wants(CompleteStatistic.class)
4    .with(Strategy.Restart);
```

Programm 7.3: Simples Beispiel, wie das Inkrementieren eines Counters in der Datenbank wieder rückgängig gemacht werden kann.

```
1  class IncrementerTask extends RollbackableTask {
2    public void task() {
3      // Inkrementiere einen Counter in der Datenbank.
4    }
5
6    public void rollback() {
7      // Dekrementiere den Counter wieder in der Datenbank.
8    }
9  }
10
11 state
12   .with(Strategy.Rollback);
```

weiterer Task implementiert wird, der etwaige Änderungen am Ende vornimmt, während die vorherigen Tasks nur definieren, welche Änderungen es gibt. Oftmals ist diese Vorgehensweise aber nicht praktikabel bzw. sehr umständlich, wofür es das Prinzip des Zurückrollens gibt. Dafür können Tasks von einer speziellen Klasse erben (RollbackableTask), um die Methode `rollback` zu implementieren, was in Programm 7.3 veranschaulicht wird. Diese kann dazu verwendet werden, etwaige Änderungen rückgängig zu machen, wenn es im Weiteren zu einem Fehler in der Ausführung gekommen ist. Oftmals ist es aber sicherer, eine Schicht zwischen einen Task und der Quelle, die verändert werden soll, einzuführen. Diese ermöglicht es dann, etwaige Änderungen zurückzusetzen, bevor diese überhaupt in der Quelle übernommen werden. Diese Schicht müsste aber zwischen den einzelnen Tasks synchronisiert werden, sofern diese auf dieselbe Quelle zugreifen und würde zu Leistungseinbußen führen, wie Transaktionen bei relationalen Datenbanken (Two-Phase Commit) [6].

7.2 Fehlertoleranz in verteilten Systemen

In verteilten Systemen ist die Fehlertoleranz nicht so trivial, wie beim Ausführen von Tasks. Die Ausfallsicherheit eines verteilen Systems ist oft gekoppelt mit dem Prinzip der verlässlichen Systeme [24, S. 355], welche unter anderem folgende Kriterien abdecken:

1. *Verfügbarkeit* ist die Eigenschaft eines Systems, zu einem gewissen Zeitpunkt funktionsfähig zu sein. Oftmals wird die Verfügbarkeit als Wahrscheinlichkeit angegeben bzw. auf vielen Hosting-Platformen für Server als Prozentsatz in Zeit gesehen

auf das Jahr. Die höchste Verfügbarkeit beschreibt also ein System, welches wahrscheinlich zu jedem Zeitpunkt funktioniert.

2. *Zuverlässigkeit* ist sehr ähnlich zur Verfügbarkeit, mit dem einen Unterschied, dass sie für ein Zeitintervall und weniger für einen Zeitpunkt gemessen wird. Ein System ist zuverlässig, wenn es fortgehend ohne Unterbrechungen funktioniert. Würde zum Beispiel ein Server einmal pro Stunde für eine Sekunde ausfallen, hätte er zwar eine Verfügbarkeit von über 99%, wäre aber nicht wirklich zuverlässig.
3. *Funktionssicherheit* beschreibt den Umstand, dass ein Ausfall des Systems schwerwiegende Folgen mit sich bringt. Leitsysteme in Kernkraftwerken, Durchflussregler in Staudämmen und ähnliche Systeme müssen ein extrem hohes Maß an Funktionssicherheit liefern [24].
4. *Wartbarkeit* bezieht sich bei der Fehlertoleranz auf die Zeit, die es benötigt bzw. wie leicht es ist, ein ausgefallenes System zu reparieren. Beispielsweise sind Systeme, die Ausfälle automatisiert erkennen und beheben, sehr wartbar und weisen damit auch oft eine hohe Verfügbarkeit auf, was die Korrelation dieser Kriterien widerspiegelt.

In diesem Zusammenhang ist es noch wichtig zu klären, was ein Ausfall eines Systems bedeutet. Oft wird ein Ausfall dadurch beschrieben, dass ein System seine Zusagen nicht mehr einhalten kann [24, S. 357]. Das bedeutet, dass gewisse Dienste nicht mehr ordnungsgemäß funktionieren, wodurch auf Anfragen keine korrekten bzw. keine Antworten geliefert werden. Aus einem Ausfall heraus können Fehler auftreten. Wenn zum Beispiel ein Worker einen Task ausführt und dieser dann ausfällt, würde der Job nie zu Ende geführt werden können, wenn der Ausfall nicht erkannt wird. Die Ursache für einen Ausfall wird als Störung bezeichnet. Ein verlässliches System zu entwickeln korreliert, sehr stark mit der Kontrolle von Störungen.

7.2.1 Fehlermodelle verteilter Systeme

Wie zuvor erwähnt, besteht ein Ausfall in einem verteiltem System darin, dass das System nicht mehr in der Lage ist, gewisse Dienste anzubieten. Ein verteiltes System kann von außen betrachtet als eine Ansammlung von Servern gesehen werden [24]. Das bedeutet, dass bei einem Ausfall des Systems die Kommunikation zwischen den Servern nicht mehr ordnungsgemäß funktioniert bzw. Server komplett ausgefallen sind. In einem verteilten System hängen Server oft von anderen ab, um Zugehörigkeiten zu trennen und das System wartbarer zu machen, wie zum Beispiel der Scheduler von seinen Workern abhängig ist. Das bedeutet, dass bei einem Ausfall des Schedulers an mehreren Stellen nach möglichen Ursachen gesucht werden muss. Ausfälle eines Servers in einem verteilten System können folgendermaßen kategorisiert werden [7]:

1. *Absturzausfälle* treten auf, wenn ein Server vorzeitig ausfällt, bis zu diesem Zeitpunkt aber einwandfrei funktioniert hat. Wichtig aufzuzeigen ist, dass der Server nach einem Absturzausfall dauerhaft ausgefallen bleibt. Typisch dafür sind zum Beispiel Speicherlecks in einer Applikation, wodurch das System früher oder später hängen bleibt und nur noch neugestartet werden kann.
2. Ein *Dienstausfall* besteht, wenn ein Server nicht mehr auf Antworten reagiert, selbst wenn die Verbindung für die Kommunikation richtig aufgebaut ist. Das

kann eintreten, wenn zum Beispiel der Server nicht auf Nachrichten wartet oder er einfach auf Anfragen keine Antworten sendet. Fehlerhafte Speicherverwaltung bzw. zu spät abbrechende Endlosschleifen können ebenfalls dazu führen.

3. Ein *zeitbedingter Ausfall* tritt auf, wenn ein Server nicht in einem gewissen Zeitrahmen antwortet. In den meisten Fällen treten solche Ausfälle auf, wenn der Server aufgrund von fehlender Leistung für eine Anfrage zu lange braucht. Entweder der Server ist zu stark ausgelastet oder der Dienst ist nicht performant genug implementiert.
4. *Ausfälle korrekter Antworten* beziehen sich meist auf Fehler im Code. Dabei antwortet ein Server auf eine Anfrage einfach falsch. Ein simples Beispiel wäre ein Dienst, der Produkte eines Elektronikherstellers nach Marken filtern kann und dabei Produkte mitliefert, die nicht zu dieser Marke gehören.
5. *Zufällige Ausfälle* sind die wohl schwierigsten Fehler. Dabei werden Ausgaben erstellt, die nie eintreten sollten, aber nicht als fehlerhaft erkannt werden können. Oft werden verteilte Systeme beschädigt, indem ein vorsätzlich falsch arbeitender Server in das System eingebunden wird, der mit anderen Servern zusammenarbeitet, um gewollt falsche Antworten zu generieren (Datenbank-Injektionen u.ä.).

7.2.2 Absturzausfälle erkennen

In der Regel treten in verteilten Systemen meist Absturz- und Dienstausfälle auf [24, S. 359], jedoch werden diese Ausfälle nicht zuvor bekannt gegeben, da sie meist abrupt auftreten und nicht vorhersehbar sind. Das bedeutet, dass das restliche System solche Fehler erkennen muss. Der Vorteil von dieser Kategorie an Ausfällen ist, dass ein Server auf einen zweiten Versuch zu kommunizieren ebenfalls nicht reagiert und nicht wie bei zufälligen Ausfällen kein eindeutiges Muster erkennbar ist. In verteilten Systemen wird oft die Annahme getroffen, dass ein Prozess in einem verteilten System den Status eines anderen Prozesses feststellen kann. Darauf bauen Mechanismen auf, wie zum Beispiel Master/Worker Monitore [12], welche mittels Timeouts ermitteln, ob ein Prozess ausgefallen ist oder nicht. Dabei sendet der Masterknoten periodisch an alle Worker eine Anfrage und wenn nach einer definierten Zeitspanne (Timeout) keine Antwort zurückkommt, kann davon ausgegangen werden, dass der Prozess nicht mehr ordnungsgemäß funktioniert und ausgefallen ist. Natürlich ist diese Annahme rein hypothetisch, weil in verteilten Systemen oft Kommunikationsprobleme auftreten können und dadurch nicht sofort auf einen Ausfall geschlossen werden kann. Außerdem ist die Ermittlung eines Wertes für den Timeout kein leichtes Verfahren und in vielen Fällen kein statisches Verfahren, sondern ein dynamisches [10]. Im Prototypen wird aufgrund dessen ein anderes Verfahren verwendet, welches mit sogenannten Herzschlägen (Heartbeats) verglichen werden kann, die bei vielen browserbasierten Websocket-Bibliotheken wie socket.io[1] zum Einsatz kommen. Dabei wird mittels Interaktionen erkannt, ob ein Prozess ausgefallen ist oder nicht. Das bedeutet, dass der Scheduler wie auch alle Worker sich gegenseitig ständig ihre Dienstverfügbarkeit mitteilen. Bekommt der Scheduler für längere Zeit keine dieser Meldungen eines Workers, kann er davon ausgehen, dass dieser ausgefallen ist. In vielen dezentralisierten System wird ein ähnliches System verwendet,

[1] https://socket.io

wobei ein Knoten diese Meldungen an seine Nachbarn sendet, die diese Meldung wiederum an ihre Nachbarn weiterleiten, wovon auch die Terminologie Gossip kommt [24].

7.2.3 Ausfälle von Workern maskieren

Erkennt der Scheduler anhand der fehlenden Meldungen, einen Ausfall eines Workers, muss er darauf reagieren können. Dieser Prozess ist im Grunde sehr einfach, da der Scheduler über alle Tasks seiner Worker Bescheid weiß. Da ein Scheduler über mehr als nur einen Worker verfügt, kann er mittels Redundanz den Ausfall sauber maskieren, indem er die Tasks des ausgefallenen Workers zurücksetzt. Dadurch geht der Fortschritt des Workers verloren, jedoch nicht der Ganze Job an sich. Die einzelnen Tasks, die der Worker hätte ausführen sollen, müssen dabei wieder in den Schedule aufgenommen werden. Dabei setzt der Scheduler einfach Tasks, die nicht vollständig ausgeführt wurden, zurück und verteilt sie auf die restlichen Worker. An dieser Stelle muss angemerkt werden, dass Fehler, die durch eine unterbrochene Ausführung zustande kommen, nicht erkannt werden können. Wenn zum Beispiel ein Task in drei Schritten Daten in eine Datenbank schreibt und der Worker nach dem zweiten Schritt ausfällt, ist nicht feststellbar, dass die ersten beiden Schritte zurückgesetzt werden sollten. Dafür wäre eine komplexes System nötig, das mit sicheren Punkten bzw. Transaktionen arbeitet.

7.2.4 Auswahlalgorithmen

Der Ausfall des zentralen Schedulers ist nicht so einfach zu maskieren, wie der Ausfall eines einzelnen Workers. Sämtliche Informationen über Tasks, die derzeit ausgeführt werden bzw. noch auf ihre Ausführung warten, Fehlerbehandlungen und Tasks die noch zurückgerollt werden müssen, liegen im lokalen Speicher des Schedulers. Außerdem baut der Scheduler über Zeit einen Cache auf, um öfter auftretende Problembeschreibungen schneller zu lösen sowie Laufzeitinformationen, die für die Lastenverteilung notwendig sind. All diese Informationen gehen bei einem Ausfall des Schedulers verloren und die Auswirkung wäre ein Ausfall des gesamten Systems. In vorherigen Kapiteln wurde dieses Problem als *single point of failure* beschrieben. Es gibt Algorithmen, die solchen Problemen entgegenwirken, diese werden allgemein als Auswahlalgorithmen (Leader Election) bezeichnet. Diese können in einem verteilten System selbstständig einen neuen Koordinator wählen, wenn der derzeitige Koordinator ausfällt [1]. Die bekanntesten werden im Folgenden erläutert, jedoch lösen sie das Problem des Informationsverlustes nicht. Aus diesem Grund wird in Abschnitt 7.2.4 noch ein weiterer Algorithmus herangezogen, der nicht direkt ein Auswahlalgorithmus ist, sich auf die Architektur des Prototypen aber sehr gut abstimmen lässt.

Bully Algorithmus

Der Bully Algorithmus ist wohl der einfachste und älteste Auswahlalgorithmus. Für diesen Algorithmus werden Knoten im verteilten Systemen, die an der Auswahl teilnehmen, mit eindeutigen aufsteigenden Zahlen gekennzeichnet. Erkennt zum Beispiel der Knoten n, dass der Koordinator ausgefallen ist, startet er eine Wahl, um den neuen Koordinator zu ermitteln. Dafür sendet er an alle Knoten mit einer höheren ID als n eine Wahlnachricht. Erhält der Knoten n keine Antwort auf diese Nachricht, ernennt er

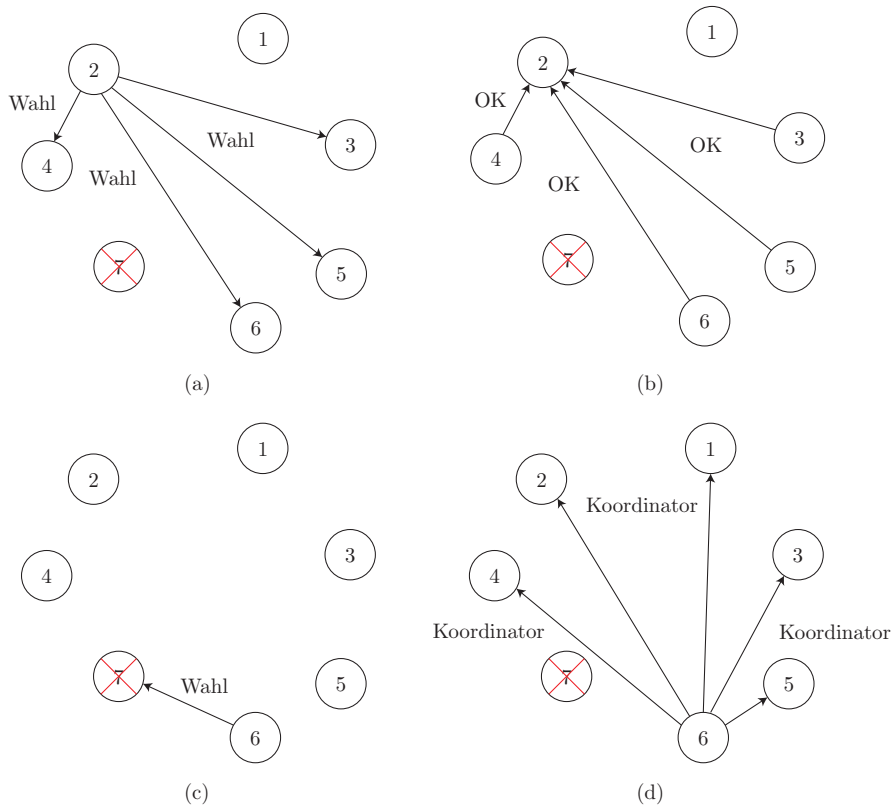


Abbildung 7.1: (a) In diesem Beispiel des Bully Algorithmus erkennt Knoten *2* den Ausfall des Koordinators *7* und startet eine Wahl. (b) Knoten *3-6* nehmen an der Wahl teil und wiederholen diesen Vorgang. (c) Knoten *3-5* erhalten ebenso Antworten auf ihre Wahlnachrichten, da Knoten mit einer höheren Zahl an der Wahl teilnehmen. Knoten *6* erhält keine Antwort, da Knoten *7* ausgefallen ist und (d) ernennt sich zum neuen Koordinator.

sich zum neuen Koordinator und teilt dies allen anderen Knoten mit. Bekommt Knoten n mindestens eine Antwort zurück, ist für ihn die Wahl vorbei und die Knoten, die auf die Nachricht geantwortet haben, wiederholen diesen Vorgang solange, bis ein Knoten keine Antworten auf die Wahlnachricht bekommt und zum neuen Koordinator gewählt wird. Nicht so trivial ist die Tatsache, dass bei diesem Algorithmus jeder Knoten alle anderen Knoten im System kennen muss und während der Wahl sehr viele Nachrichten ausgetauscht werden müssen. Das führt dazu, dass der Bully Algorithmus nicht sehr skalierbar ist. Abbildung 7.1 demonstriert den Ablauf des Bully Algorithmus.

Improved Bully Algorithm

Diese verbesserte Version des Bully Algorithmus versucht die Schwachstellen des originalen Algorithmus aus Abschnitt 7.2.4 zu bereinigen. Dabei wird nicht erst mit einer Wahl gestartet, wenn der Koordinator ausfällt, sondern es wird bereits im Vorfeld ein neuer gewählt. Dazu sammelt der Koordinator während der Laufzeit Informationen über das System, ähnlich wie bei der Lastenverteilung, und wählt einen Kandidaten aus. Erkennt

ein Knoten den Ausfall des Koordinators, vergleicht er seine ID mit dem Kandidaten. Ist diese höher als der Kandidat, startet er den Bully Algorithmus. Dadurch wird die Anzahl der Knoten, die an der Wahl teilnehmen, stark reduziert, wodurch weniger Nachrichten gesendet werden müssen. Das Problem, dass jeder Knoten jeden anderen kennen muss, wird dadurch jedoch nicht gelöst. Außerdem ist das Sammeln der nötigen Informationen zur Wahl eines Kandidaten nicht trivial und benötigt eine große Datenbank [1].

Modified Election Algorithmus

Ebenso auf der Basis des Bully Algorithmus wurde der Modified Election Algorithmus entwickelt. Dieser reduziert die Anzahl der Nachrichten, ohne dafür einen Kandidaten während der Laufzeit wählen zu müssen. Dafür sendet ein Knoten n ebenfalls eine Wahlnachricht an alle Knoten mit einer höheren ID. Diese Antworten direkt mit ihrer eigenen ID. Erhält Knoten n Antworten zurück, sammelt er die IDs aus den Antworten zusammen und sucht die höchste aus. Mit dieser Information sendet er dem Knoten mit dieser ID eine Nachricht, um ihn zum neuen Koordinator zu wählen. Diese Nachricht verteilt der neue Koordinator an alle Knoten im System, um die Wahl abzuschließen. Die fehlende Skalierbarkeit des Bully Algorithmus wird auch durch den Modified Election Algorithmus nicht gelöst, aber die Anzahl der benötigten Nachrichten zu Auswahl eines neuen Koordinators wird drastisch gesenkt. Allerdings ist der Modified Election Algorithmus zeitgebunden. Wird zum Beispiel ein neuer Knoten m im System registriert, nachdem Knoten n die Wahl gestartet hat, gibt es für Knoten m keine Möglichkeit, an der Wahl teilzunehmen, auch wenn er die höchste ID im System hätte. Abbildung 7.2 zeigt den Ablauf des Modified Election Algorithmus.

Ring Algorithmus

Für den Ring Algorithmus werden die Knoten logisch bzw. physikalisch in einem Ring angeordnet. Logisch bedeutet in diesem Zusammenhang, dass zwei Knoten, die im Ring hintereinander gereiht sind, physikalisch weit auseinander liegen können. Jeder Knoten kennt seinen Nachfolger (successor) und kann diesen beim Austauschen von Nachrichten ggf. überspringen. Wenn ein Knoten in diesem Ring erkennt, dass der Koordinator ausgefallen ist, erstellt er eine Wahlnachricht, in die er seine ID legt und an seinen Nachfolger schickt. Dieser legt ebenfalls seine ID in die Nachricht und sendet sie an seinen Nachfolger. Dieser Mechanismus wiederholt sich solange, bis die Nachricht zum Knoten zurückkommt, der die Wahl gestartet hat. Dieser erkennt, ob es sich um seine eigene Wahlnachricht handelt, wenn seine eigene ID in der Nachricht vorhanden ist. Aus der Liste wird die höchste ID gewählt und in eine Koordinatornachricht gelegt, die ebenfalls wieder im Ring weitergereicht wird. Wenn diese einmal komplett zirkuliert ist, wird sie gelöscht und die Wahl damit abgeschlossen. Die Anzahl der Nachrichten für den Ring Algorithmus ist im Vergleich zu den anderen Algorithmen sehr gering [1]. Wie Knoten in einem verteilen System logisch in Form eines Ringes angeordnet werden können, wird in Abschnitt 8.2.1 anhand von einer verteilten Hash Tabelle (Chord) genauer erläutert.

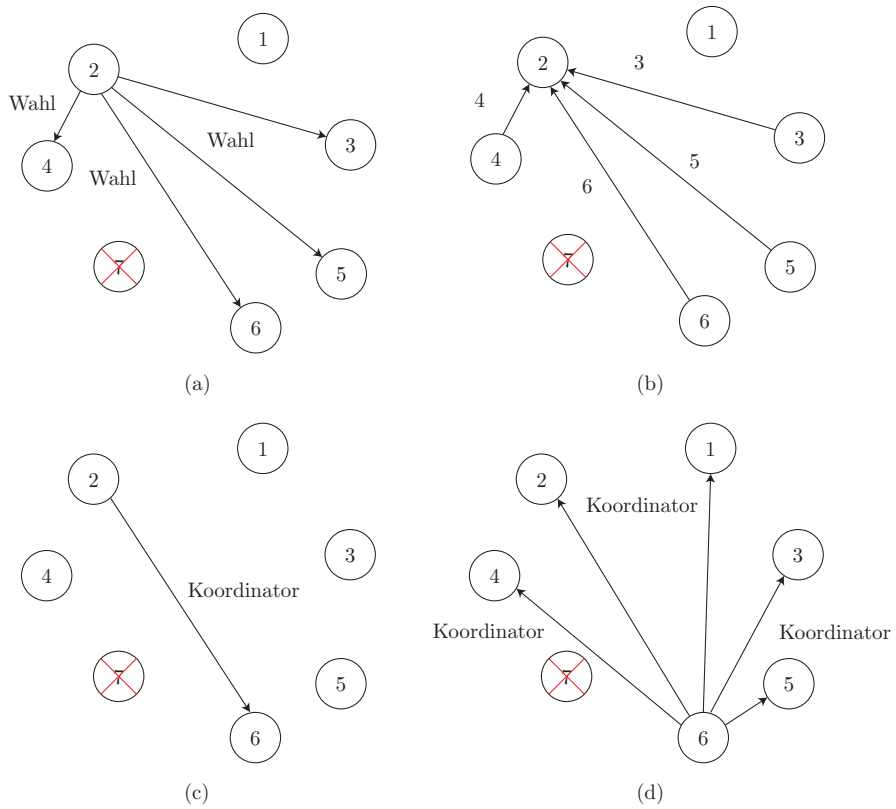


Abbildung 7.2: (a) In diesem Beispiel des Modified Election Algorithmus erkennt wieder Knoten *2* den Ausfall des Koordinators *7* und startet eine Wahl. (b) Knoten *3-6* nehmen an der Wahl teil und senden ihre ID an Knoten *2*. (c) Knoten *2* sucht die höchste ID aus den Antworten und wählt Knoten *6* zum Koordinator. (d) Knoten *6* gibt allen anderen Knoten von dem Ergebnis Bescheid und schließt die Wahl ab.

Ausfälle maskieren durch Redundanz

Ausfälle einer Koordinators müssen und können nicht immer mittels eines Auswahlalgorithmus gelöst werden. Wie zuvor erwähnt könnte es sein, dass ein Koordinator über die Laufzeit einen Cache aufbaut bzw. wichtige Informationen sammelt. In der Informatik wird ein solches Verhalten als kalter Start bezeichnet, da das System anfangs sehr träge und langsam ist, bis der Cache die wichtigsten Informationen beinhaltet bzw. die benötigten Informationen gesammelt worden sind. Nur durch den Einsatz von Auswahlalgorithmen kann ein kalter Start nicht verhindert werden. Durch den Einsatz von Redundanz können kalte Starts leicht umgangen werden, wobei die Informationen, die einen kalten Start verursachen, auf mehrere Knoten kopiert werden, damit diese nach der Auswahl eines neuen Koordinators bereits zur Verfügung stehen. *Palantir*[2] stellte auf der We Are Developers[3] Konferenz 2017 im Rahmen ihrer verteilen sicheren Suche ein Verfahren vor, um schnell einen Koordinator zu wählen und einen kalten Start mittels Redundanz zu verhindern. Bei diesem Verfahren kann nicht wie bei herkömmlichen

[2] https://www.palantir.com/
[3] https://www.wearedevelopers.com

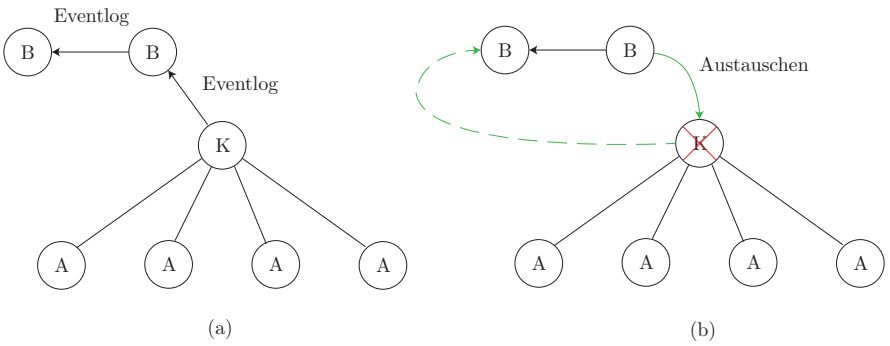


Abbildung 7.3: (a) Genereller Aufbau des Systems basierend auf dem Verfahren von Palantir. Anhand des Eventlogs des Koordinators k werden die Backupknoten B aktualisiert und mit dem Koordinator synchronisiert. (b) Fällt der Koordinator aus, übernimmt der erste Backupknoten in der Warteschlange seinen Platz. Sollte sich der alte Koordinator wieder erholen, sei es manuell oder automatisch, reiht er sich in der Warteschlange ein.

Auswahlalgorithmen jeder Knoten zum Koordinator gewählt werden, sondern nur dedizierte Knoten. Diese registrieren sich in einer Warteschlange und bilden dadurch eine Liste von Kandidaten. Der Koordinator selbst loggt wichtige Informationen, die einen kalten Start produzieren würden, damit sie vom ersten Knoten in der Warteschlange ausgelesen werden können, um diesen mit dem Koordinator zu synchronisieren. Dieser Mechanismus wiederholt sich, wie beim Ring-Algorithmus, womit alle Knoten in der Warteschlange synchron mit ihrem Vorgänger sind. Das bedeutet aber auch, dass Knoten, die in der Warteschlange weiter hinten liegen, nicht synchron mit dem Koordinator selbst sind. Fällt der Koordinator aus, wird sein Platz vom ersten Knoten in der Warteschlange eingenommen. Dieser teilt allen Knoten mit, dass sich der Koordinator geändert hat. Abbildung 7.3 zeigt den grundsätzlichen Aufbau dieses Verfahrens. Der Grund, warum dieses Verfahren in dieser Arbeit aufgezeigt wird, liegt an der Kompatibilität mit dem Prototypen. Der Mechanismus, um wichtige Informationen zu loggen, lässt sich sehr einfach mit dem event-basierten Aufbau des Prototypen verbinden. Da jeder erfolgreich bearbeitete Task, jeder neue Task, jede Anfrage an das System und vieles mehr durch ein Event repräsentiert wird, können diese Informationen zentral geloggt werden und somit der nächste Knoten in der Warteschlange mit dem Scheduler synchronisiert werden.

Kapitel 8

Dezentralisierung eines zentralisierten Systems

Wie zuvor erwähnt, wurde für den Prototypen eine recht einfache, zentralisierte Architektur des Systems gewählt, um als Proof of Concept für die Lastenverteilung mittels Graphenplanung zu dienen. Aus dem vorherigen Kapitel erschließen sich jedoch einige Schwachstellen einer solchen Architektur. Dieses Kapitel erläutert jene Schwachstellen und liefert einen Lösungsansatz, durch das Einbringen einer weiteren Schicht mittels verteilten Hash-Tabellen.

8.1 Schwachstellen der zentralisierten Architektur

Der große Vorteil verteilter Systeme besteht in ihrer Skalierbarkeit. Das bedeutet, dass das System durch das Hinzufügen von weiteren Knoten (Hardware) mehr Lasten verarbeiten bzw. mehr Anfragen beantworten kann, je nach Art der Applikation. Ab einer bestimmten Größe und Architektur des verteilten Systems, ist das Verhältnis von Hardware zu Leistung nicht mehr 1:1, da pro Knoten ein gewisser Overhead mit eingerechnet werden muss. Man spricht von einem skalierbaren System, wenn dieser Overhead kaum spürbar ist. Bei einer zentralisierten Architektur besteht dieser Overhead aus den Nachrichten, die der zentrale Koordinator verarbeiten muss. Auf Grund dessen, dass immer nur ein Koordinator existiert, ist diese Anzahl der Nachrichten durch die Leistung des Koordinators begrenzt. Das führt dazu, dass eine zentralisierte Architektur nicht skalierbar ist und ab einer bestimmten Größe ihre Kapazität erreicht. Der Prototyp erreichte zum Beispiel auf einem Ubuntu VPS mit zwei Kernen zu je 2,40 GHz und 4GB RAM eine Kapazität von 30000 Nachrichten pro Sekunde. Aus diesem Grund ist die Anzahl der Worker begrenzt, da der Koordinator nicht mehr mithalten könnte. Der zweite Schwachpunkt liegt, wie in Abschnitt 7.2.4 beschrieben, in der Fehlertoleranz. Bei einem Ausfall des zentralen Koordinators können wie im Prototypen wichtige Informationen verloren gehen. Durch Redundanz der Daten und Austauschen des Koordinators kann dem entgegengewirkt werden, dabei gehen aber wertvolle Sekunden verloren, bis das System wieder verfügbar ist. Außerdem müssen auch hier noch weitere Vorkehrungen getroffen werden, denn es kann immer passieren, dass auch Backupknoten ausfallen und dadurch das System zum Stillstand kommt.

Eine einfache Lösung für diese Schwachstellen gibt es nicht. Es gibt jedoch die Möglichkeit, eine zentrale Architektur, wie die des Prototypen, zu dezentralisieren. In dieser Arbeit wird ein Verfahren herangezogen, welches auf den Prototypen angewendet werden kann und das System mittels verteilter Hash-Tabellen dezentralisiert [20], wodurch eine höhere Skalierbarkeit sowie Verfügbarkeit gewährleistet wird.

8.2 Verteilte Hash-Tabellen

Verteilte Hash-Tabellen (Distributed Hash Table, DHT), sind, wie ihr Name bereits erahnen lässt, nach außen hin wie eine normale Hash-Tabelle. Sie bieten dieselbe Schnittstelle an, wobei mit `put(key, value)` Daten in die Tabelle zu einem Schlüssel abgespeichert und mit `get(key)` wieder ausgelesen werden können. Der große Unterschied zu einer herkömmlichen Hash-Tabelle ist der, dass die Daten dabei auf verschiedene Knoten verteilt werden. Dabei gibt es viele verschiedene Implementierungen, wie die einzelnen Knoten einer DHT logisch angeordnet werden, um die Daten gleichmäßig zu verteilen. Chord ist in vielen Literaturen die repräsentative Implementierung einer DHT und wird daher auch in dieser Arbeit herangezogen, wobei zusätzlich die bekanntesten DHT-Implementierungen am Ende dieses Abschnittes kurz erwähnt werden, um etwaige Unterschiede aufzuzeigen bezüglich Lookup-Protokollen und Ähnlichem.

8.2.1 Chord

Chord verwendet einen m-Bit-Bezeichnerraum [24, S. 216], um Knoten sowie Daten einen Bezeichner, oder auch Schlüssel genannt, zuzuweisen und in der DHT zu hinterlegen. Diese Daten können sich von Anwendung zu Anwendung unterscheiden. Im Prototypen wären solche Daten Tasks und deren Zustände. In anderen Anwendungen könnten es Dateien oder Prozesse sein. Die Anzahl m der Bits für einen Schlüssel beträgt in den meisten Fällen 128 oder 160 und hängt von der Hashfunktion ab, die verwendet wird (MD5, SHA-1, usw.). Ein Datensatz mit dem Schlüssel s fällt in den Einflussbereich des Knotens mit dem kleinsten nächsten Schlüssel t, wobei gelten muss $t \geq s$. Dieser Knoten k wird als Nachfolger (Successor) von s bezeichnet und im Weiteren mit $succ(s)$ abgekürzt [24, S. 216]. Das bedeutet, dass die Schlüssel für Datensätze und Knoten in der DHT mittels derselben Hashfunktion berechnet werden. Bevor besprochen werden kann, wie der Datensatz zu einem zugehörigem Schlüssel nachgeschlagen werden kann, ist es wichtig zu erläutern, wie Schlüssel in Chord gereiht werden. In Chord kennt jeder Knoten seinen *succ*, wodurch die Knoten einen logischen Ring bilden. Wird ein Datensatz mit dem Schlüssel s mit `put` in der DHT abgelegt, wird er auf dem Knoten $succ(s)$ abgespeichert und landet auf dem Ring. Abbildung 8.1 zeigt den logischen Aufbau einer Chord DHT mit einem Schlüsselraum von drei Bits. Hinzuzufügen ist, dass ein Datensatz den selben Schlüssel wie ein Knoten haben kann, wobei dieser Knoten der *succ* des Datensatzes wäre. Um Chord etwas zu vereinfachen, wird in den Beispielen ein Schlüssel jeweils nur einmal vergeben.

Die hauptsächliche Aufgabe einer DHT ist es, möglichst effizient zu einem Schlüssel s den Nachfolger $succ(s)$ zu finden, der für diesen Schlüssel zuständig ist [24, S. 216]. Ein naiver Ansatz wäre es zu denken, dass ein Knoten k seinen Nachfolger $succ(k)$ und seinen Vorgänger $pred(k)$ kennt. Damit kann linear in der DHT nachgeschlagen werden,

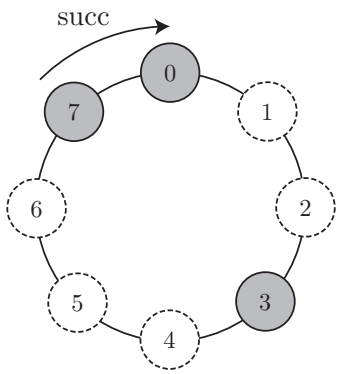


Abbildung 8.1: Aufbau einer Chord DHT mit einem Schlüsselraum von drei Bits. Graue Knoten repräsentieren Knoten der DHT, während weiße Knoten Datensätze darstellen.

wobei ein Knoten einen Schlüssel, für den er nicht zuständig ist, einfach an seinen Nachfolger bzw. Vorgänger weiterleitet. Im Fall $pred(k) < s \leq k$ ist der Knoten k selbst für den Schlüssel s zuständig und gibt seine eigene Adresse und Port an den Prozess zurück, der die Anfrage gestartet hat, damit der Datensatz im Weiteren vom Knoten k ausgelesen werden kann. Aus Leistungsgründen wird in Chord nicht linear nachgeschlagen, stattdessen baut jeder Knoten eine sogenannte Fingertabelle mit maximal m Einträgen auf [24, S. 217]. Um diese aufzubauen wird folgende Formel [20] verwendet, wobei FT_k die Fingertabelle von Knoten k bezeichnet und $FT_k[i]$ den Eintrag der Fingertabelle an Index i kennzeichnet, dabei wird bei Index $i = 1$ gestartet

$$FT_k[i] = succ((k + 2^{i-1}) \bmod 2^m).$$

Wichtig anzumerken ist, dass nicht zwangsläufig auf den Knoten verwiesen wird, der den Schlüssel $(k + 2^{i-1}) \bmod 2^m$ besitzt, sondern, falls dieser nicht vorhanden ist, auf dessen Nachfolger. Dadurch wird gewährleistet, dass ein Eintrag in einer Fingertabelle immer auf einen Knoten zeigt, der bereits im Schlüsselraum vorhanden ist. Die einzelnen Einträge steigen mit zunehmendem Index exponentiell an, wodurch auch bei einem sehr großen Schlüsselraum nicht viele Sprünge (Hops) benötigt werden. Um in Chord einen Schlüssel s nachzuschlagen, leitet Knoten k die Anfrage an den Knoten l mit dem Index j in der Fingertabelle von k weiter, wenn:

$$l = FT_k[j] \leq s < FT_k[j + 1]$$

bzw. $l = FT_k[1]$ wenn $k \leq s \leq FT_k[1]$ [24, S. 217]. Um dieses Verfahren zu veranschaulichen, zeigt Abbildung 8.2 eine Chord DHT mit einem Schlüsselraum von fünf Bits und neun registrierten Knoten an den Schlüsseln 1, 4, 9, 11, 14, 18, 20, 21 und 28 (Beispiel aus [24, S. 217]). Im ersten Beispiel, markiert durch die durchgehenden Pfeile, versucht Knoten 1 den Schlüssel $s = 26$ aufzulösen. Knoten 1 erkennt, dass der Schlüssel größer als alle Einträge in seiner eigenen Fingertabelle ist und leitet damit die Anfrage an den letzten Eintrag $FT[5] = 18$ weiter. Knoten 18 wählt laut dem Verfahren Knoten 20 aus, da $FT_{18}[2] < s \leq FT_{18}[3]$. Anschließend delegiert Knoten 20 die Anfrage an Knoten 21 und dieser schließlich an Knoten 28, der für den Schlüssel $s = 26$ zuständig ist.

Knoten 28 gibt seine Adresse und Port an Knoten 1 zurück, womit das Nachschlagen abgeschlossen wird. Danach kann Knoten 1 eine Verbindung zu Knoten 28 aufbauen und die benötigten Daten abfragen. Die gestrichelten Pfeile in Abbildung 8.2 zeigen ein weiteres Beispiel, in dem Knoten 28 den Schlüssel $s = 12$ auflöst.

Knoten können recht einfach in das System hinzugefügt werden. Angenommen, ein Knoten u möchte in das System aufgenommen werden. Dazu nimmt er einfach Kontakt zu einem Knoten im System auf und lässt den Schlüssel $succ(u + 1)$ auflösen. Danach kann sich Knoten u in den Ring einfügen. Die Schwierigkeit liegt darin, die einzelnen Fingertabellen der Knoten aktuell zu halten. Dafür ist es wichtig, dass für jeden Knoten k zutrifft, dass $FT_k[1]$ auf den nächsten Knoten im Ring verweist [24, S. 218]. Dazu nimmt jeder Knoten k regelmäßig Kontakt mit seinem Nachfolger auf und prüft, ob zutrifft, dass $k = pred(succ(k + 1))$. Stimmen diese nicht überein, weiß k, dass ein neuer Knoten im System hinzugefügt worden ist und aktualisiert seine Fingertabelle [24, S. 218]. Manche System unterscheiden sich hierbei, dass diese Prozedur erst im Falle eines Fehler ausgeführt wird und dadurch das System träge aktuell hält.

8.2.2 Optimierungen nach Netzwerknähe

Eines der Probleme, die bei DHTs auftreten können, liegt darin, dass Anfragen im Zuge einer Schlüsselauflösung quer durch das Netzwerk gehen können. Ist die DHT geographisch sehr weit verteilt, kann es auftreten, dass ein Schritt über die Fingertabelle weite Strecken zurücklegen muss. Das endet darin, dass Anfragen länger dauern, als sie eigentlich müssten. Um dieses Verhalten zu verhindern, gibt es mehrere Strategien, wodurch dies meist durch die Berechnung der Schlüssel gelöst werden kann, um geographisch naheliegende Knoten zum Beispiel auch im Ring von Chord derartig abzulegen. Diese Strategie kommt bei der topologiebasierten Zuweisung zum Einsatz [24, S. 219]. Dieses Vorgehen kann jedoch andere Probleme mit sich bringen. Werden nahe beieinanderliegende Knoten im zweidimensionalen Ring ebenso als Nachbarn abgelegt, bilden kleine Netzwerke wie zum Beispiel eines Unternehmens einen kleinen Bereich am Ring ab. Beim Ausfallen dieses Netzwerkes, würde ein großer zusammenhängender Teil des Ringes verloren gehen, was für das Auflösen von Schlüsseln sehr große Konsequenzen mit sich trägt, da womöglich dieser Teil des Ringes nicht mehr übersprungen werden kann [24, S. 219]. Sind die Knoten des Netzwerkes jedoch weit am Ring verstreut, hat ein Ausfall keine großen Auswirkungen. Die Strategie des Umgebungsroutings baut auf der Idee der Redundanz auf. Dabei merkt sich ein Knoten k nicht nur seinen direkten Nachfolger $succ(k)$, sondern s viele Nachfolger [24, S. 219]. Dieses Verhalten kann auf jede Fingertabelle erweitert werden, womit eine zweidimensionale Tabelle erstellt wird und jeder Eintrag eine Liste von r Knoten darstellt. Das bedeutet, dass ein Eintrag in der Fingertabelle $FT_k[i]$ des Knoten k auf den Bereich $[p+2^{i-1}, p+2^i-1]$ verweist. Dadurch können Ausfälle besser abgedeckt werden, da mehrere Routen für das Auflösen zur Verfügung stehen. Der Nachteil liegt im aufwendigeren Aktualisieren der Fingertabellen bei Ausfällen und neu hinzugekommenen Knoten.

8.2.3 Weitere DHTs

Chord zählt zu einem der einfachsten Systeme einer verteilten Hash-Tabelle. Die Ringtopologie ist einfach zu konstruieren und überschaubar in ihrer Komplexität. Andere

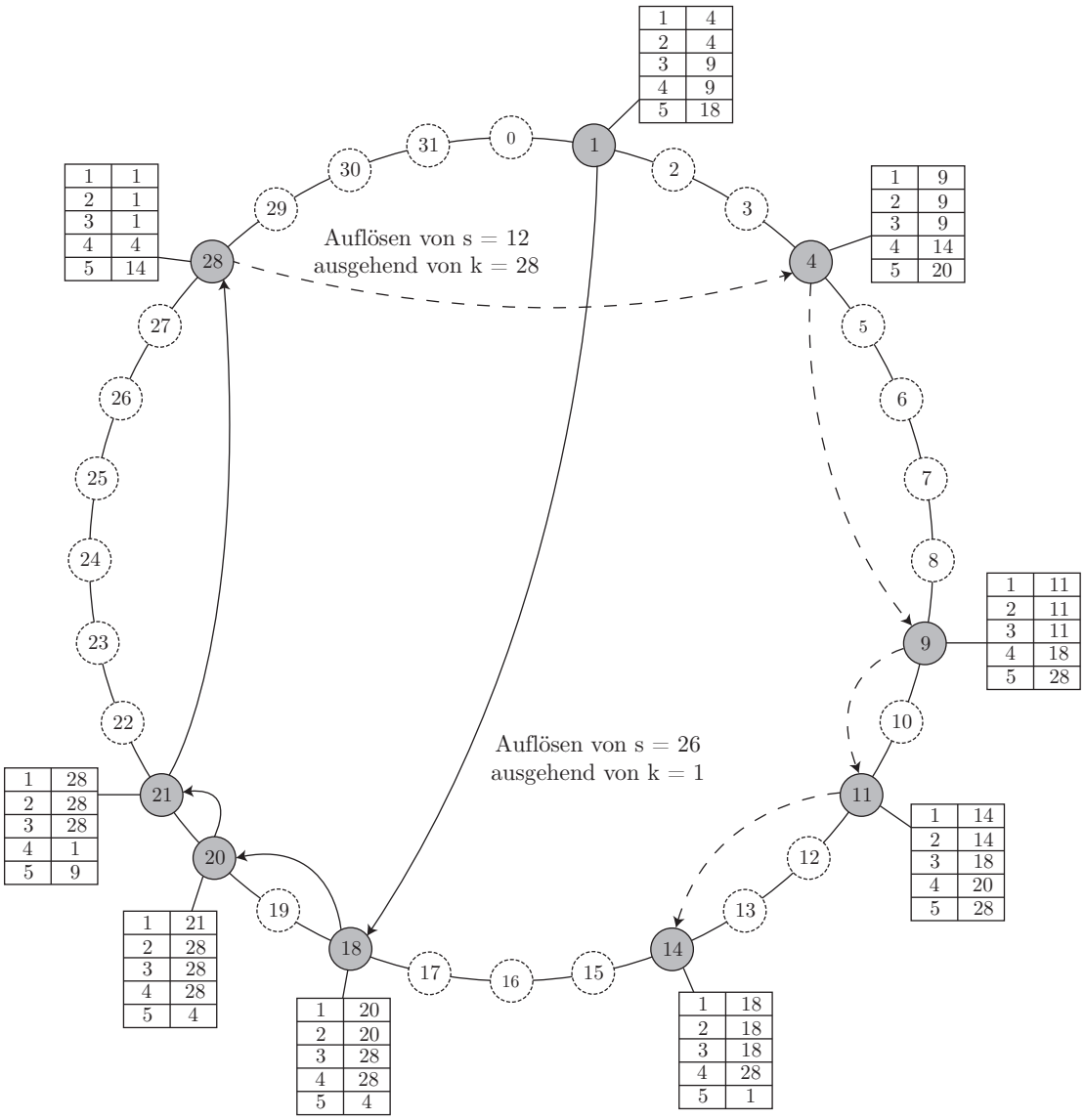


Abbildung 8.2: Nachschlagen von Schlüssel $s = 26$ ausgehend von Knoten $k = 1$ und Schlüssel $s = 12$ ausgehend von Knoten $k = 28$ in einer Chord DHT laut [24, S. 217].

Implementierungen wie Pastry und Symphony verwenden ebenfalls die Ringtopologie, jedoch unterscheiden sich die Fingertabellen sehr stark. In Symphony zum Beispiel werden die Einträge in der Fingertabelle als Link bezeichnet. Dabei wird zuerst ein Link zum unmittelbaren Nachfolger $succ(k)$ eines Knoten k aufgebaut. Danach werden sogenannte Long-Links erstellt, welche bi- aber auch unidirektional sein können. Dafür werden Zufallszahlen mit einer Wahrscheinlichkeits-Funktion (Propability Distribution Function, PDF) berechnet, welche approximiert, wie viele Knoten im Netzwerk existieren [17]. Die Anzahl der Knoten im System wird mittels Kreissegmenten geschätzt, wobei das genaue Verfahren in dieser Arbeit nicht erläutert wird. Mithilfe dieser Zahlen werden die einzelnen Long-Links aufgebaut, um auch größere Sprünge beim Auflösen

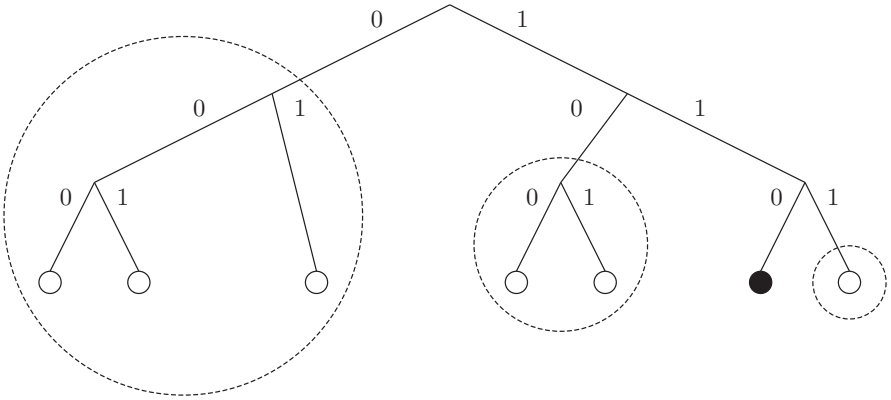


Abbildung 8.3: Sturktur von Kademlia ausgehend von dem Knoten mit dem Schlüssel 110. Die gestrichelten Kreise repräsentieren die einzelnen Regionen, die mit der XOR Metrik berechnet werden. Die Blätter des Baumes sind die Knoten des Systems.

eines Schlüssels zu ermöglichen.

Es gibt aber auch Implementierungen, die nicht auf der Ringtopologie aufbauen, sondern einen hierarchischen Ansatz verfolgen. Kademlia verwendet eine logische Baumtopologie und wird bei einem sehr bekannten Peer to Peer System eingesetzt, BitTorrent[1] [27]. Um eine Baumstruktur zu bilden, verwendet Kademlia die XOR-Metrik, um Distanzen zwischen Schlüsseln zu berechnen (Anzahl der unterschiedlichen Bits). Knoten die durch diese Metrik nicht weit voneinander entfernt sind, bilden gemeinsame Regionen in der Baumstruktur [18]. Ein Knoten k hält eine Liste von sogenannten k-buckets, welche wiederum Listen mit k Einträgen sind. Jeder k-bucket enthält dadurch k viele Links zu einer bestimmten Region mittels der XOR-Metrik. Dadurch sind wiederum kurze und längere Sprünge beim Auflösen eines Schlüssels möglich. Diese Struktur wird in Abbildung 8.3 aufgezeigt.

8.3 Dezentralisierung mittels DHTs

Wie zuvor erwähnt, bildet der Koordinator in einem zentralisierten System den Flaschenhals bezüglich der Skalierbarkeit und auch der Verfügbarkeit der Ressourcen (CPU, Daten, ...). Um dieses Problem zu lösen, kann ein solches System dezentralisiert werden. Im Falle des Prototypen würde das heißen, mehrere Scheduler im System zu erstellen, die für eine gewisse Anzahl von Workern zuständig sind. Zu beachten sind jedoch zwei Punkte eines solchen Ansatzes. Zum einen lösen mehrere Scheduler nicht das Problem des Datenverlustes im Falle eines Ausfalles und außerdem spielt die Lastenverteilung auf die einzelnen Scheduler eine große Rolle. In diesem Abschnitt wird kurz beschrieben, wie ein System aufgebaut werden kann, welches multiple Scheduler zulässt und wie die zuvor erwähnten Probleme gelöst werden können.

[1] http://www.bittorrent.org

8.3.1 Multiple Scheduler

Um ein System zu entwickeln, welches mehrere Scheduler unterstützt, kann eine weitere Schicht in die Architektur eingeführt werden. Diese benötigt keine weiteren Knoten, sondern verwendet die einzelnen Scheduler, um eine DHT zu bilden. Das bedeutet, dass die Scheduler die Knoten bilden, auf denen Daten in der DHT abgelegt und ausgelesen werden können. Außerdem implementieren sie die nötigen Protokolle, um Schlüssel aufzulösen und zu definieren, welcher Scheduler für welche Daten zuständig ist laut Chord [20]. Dadurch bildet die DHT eine globale Anlaufstelle, um Problemlösungen von einem Scheduler lösen zu lassen, wobei abstrahiert wird, dass mehrere Scheduler im System vorhanden sind. Im Vorfeld muss jedoch definiert werden, welche Daten im Falle des Prototypen in der DHT abgespeichert werden. Im Grunde werden ganze Jobs abgespeichert, was dazu führt, dass immer genau ein Scheduler für einen gesamten Job zuständig ist. Das hat den Grund, dass das Verteilen der einzelnen Tasks zu viele Leseoperationen [5] mit sich bringen würde, wenn ein Scheduler berechnet, ob ein Task bereits ausführbar ist oder nicht. Dabei müsste für jeden voraussetzenden Task eine Leseoperation ausgeführt werden, da der Task bei einem anderen Scheduler liegen kann. An dieser Stelle sei noch angemerkt, dass Tasks ebenfalls in der DHT abgespeichert werden, deren Schlüssel jedoch auf dieselbe Art berechnet wird, wie die des Jobs, zu dem sie gehören. Das führt dazu, dass ein Task am selben Scheduler gespeichert wird. Der Grund, warum Tasks ebenfalls in der DHT abgelegt werden, wird in Abschnitt 8.3.3 weiter erläutert.

8.3.2 Lastenverteilung zwischen den Schedulern

Der große Vorteil von DHTs ist ihre Skalierbarkeit und die Eigenschaft, auf ausfallende und neu hinzukommende Knoten zu reagieren und sich selbst zu verwalten. Diese Punkte und auch die Performanz einer DHT hängen sehr stark von einer gleichmäßigen Verteilung der Lasten bzw. Daten ab [4, 22]. Dadurch, dass Implementierungen wie Chord zyklische Hashfunktionen verwenden, wird die Last von Grund auf sehr gleichmäßig verteilt. Es gibt jedoch Möglichkeiten und Algorithmen, diesen Prozess weiter zu verbessern.

Power of Two Choices

Ein einfaches Konzept besteht darin, verschiedene Hashfunktionen gleichzeitig zu verwenden. Dieses wird oft als *Power of Two Choices* [19] bezeichnet. Wird zum Beispiel ein neuer Wert in der DHT abgelegt, werden zuerst mehrere Schlüssel mittels den verschiedenen Hashfunktionen berechnet. Die einzelnen Schlüssel werden aufgelöst und der Wert auf dem Knoten gespeichert, der derzeit am wenigsten ausgelastet ist. Der Nachteil liegt darin, dass der Prozess, welcher ursprünglich nur einmal ausgeführt wird, für jede Hashfunktion abgearbeitet werden muss. Das gilt im Zuge dessen für Schreib- und Lesezugriffe. Für Daten, die nur einmalig abgelegt und kaum verändert werden, ist dieses Verfahren jedoch optimal. Für den Prototypen wären die redundanten Zugriffe jedoch ein zu großer Overhead, da sich die Daten der DHT sehr rasch und oft ändern.

Virtuelle Server

Für den Prototypen würde sich das Konzept der virtuellen Server anbieten. Dabei wird die DHT in Partitionen eingeteilt, indem ein physikalischer Knoten sich als mehrere logische bzw. virtuelle Server ausgibt [22]. Für die DHT ist jeder virtuelle Server wie ein eigenständiger Knoten. Wie zuvor erwähnt ist in Chord jeder Knoten für einen Bereich auf dem Ring zuständig. Dabei kann es passieren, dass sich viele Daten in gewissen Regionen des Ringes ansammeln und in manchen Regionen wiederum nur sehr wenige Daten [4]. Bei dem Konzept der virtuellen Server wird versucht, dass diese Regionen ausgeglichen werden. Das bedeutet, dass ein physikalischer Knoten für Regionen zuständig ist, in denen viele Daten liegen und gleichzeitig für Regionen mit wenig Daten. Um zu ermöglichen, dass Daten, welche sich oft ändern, gleichmäßig verteilt werden können, müssen zusätzliche Algorithmen für das Migrieren von virtuellen Servern von Knoten zu Knoten eingesetzt werden, wie zum Beispiel *k-choices* und *Many-to-Many* [4]. Diese genauer zu erläutern, würde den Rahmen der Arbeit jedoch sprengen, wesegen sie in diesem Zuge nur erwähnt werden sollen.

8.3.3 Maskieren von Ausfällen mittels Redundanz

Bilden die einzelnen Scheduler des Systems eine DHT, wären Auswahlalgorithmen bzw. das Maskieren eines Ausfalls durch Backupknoten obsolet. Für eine DHT ist es ein Leichtes, Daten redundant abzuspeichern, um den Ausfall eines Knotens zu maskieren. Dabei werden Daten auf n weitere Knoten, meistens Nachfolgern, abgespeichert. Mittels dieser Technik können Schlüssel trotzdem auf herkömmliche Weise aufgelöst werden, wenn ein Knoten ausgefallen ist. Zu beachten ist jedoch der Overhead, der beim Abspeichern von Daten dazukommt, welcher abhängig von der gewählten Strategie der Redundanz ist [5]. Dabei gibt es zwei unterschiedliche Strategien, welche in der Datenredundanz oft vorkommen. Wie zuvor erwähnt gibt es die Möglichkeit, einen kompletten Datensatz auf n weiteren Knoten abzuspeichern. Dabei muss beim Lesen eines Datensatzes mindestens einer der n Knoten verfügbar sein. Beim Speichern und vor allem beim Aktualisieren der Daten müssen jedoch alle der n Knoten verfügbar sein, da es ansonsten zu Inkonsistenzen kommen kann. Eine weitere Strategie basiert auf dem Prinzip, welches in Raidsystemen verwendet wird und in Literaturen als „k *out-of* n“ bezeichnet wird. Dabei wird ein Datensatz in k Teile aufgespalten, wobei zusätzlich $n - k$ redundante Stücke hinzugefügt werden [5]. Das bedeutet, dass im System n Teile des Datensatzes gespeichert werden und nur k Teile benötigt werden, um den Datensatz wiederherstellen zu können. In [5] wird aufgezeigt, dass in einem k *out-of* n System die Möglichkeit besteht, dass ein Aktualisieren der Daten dieselbe Verfügbarkeit haben kann, wie ein Lesezugriff. Dieses Verhalten ist im Prototypen von äußerster Wichtigkeit, da sich Daten sehr häufig ändern. Wie zuvor erwähnt, würden dabei Tasks in der DHT abgespeichert werden. Durch die Redundanz wüssten also mehrere Scheduler über den Status eines Tasks Bescheid, was bedeutet, dass im Falle eines Ausfalles ein Scheduler den anderen ersetzen kann. Die einzelnen Worker, für die ein Scheduler zuständig ist, würden jedoch dennoch verloren gehen, da ein anderer Scheduler nicht die Kapazitäten hätte, diese aufzunehmen. Für das System an sich wäre der Ausfall eines Schedulers jedoch verkraftbar, da davon ausgegangen werden kann, dass dieser wiederhergestellt wird, sei es nun manuell oder automatisch.

Kapitel 9

Evaluierung und Verbesserungen

In den vorherigen Kapiteln wurde beschrieben, wie anhand von Planungsgraphen ein Ausführungsplan erstellt und dieser in einem verteilten System gegebenenfalls parallel abgearbeitet werden kann. Es gibt jedoch Fälle, in denen dieses Verfahren nicht geeignet ist und es existieren auch einige Punkte bezüglich der Implementierung, die kritisch betrachtet werden müssen. Diese Kapitel zeigt die besagten Punkte auf und es wird versucht, auf mögliche Verbesserungen hinzuweisen.

9.1 Instanzieren von Aktionen

In Abschnitt 3.1 wurde erwähnt, dass Aktionen in der Graphenplanung mittels Objekten aus dem Definitionsbereich voll instanziert werden können. Ist zum Beispiel eine Aktion $Koenig(p)$ gegeben, die definiert, dass eine Person p ein König ist und das Objekt $David$ eine Person, so kann die Aktion voll instanziert werden und ergibt $Koenig(David)$. Dieses Verhalten ist im Prototypen mit Annotationen abbildbar, da mittels Beans deklariert wird, welche Voraussetzungen gegeben sein müssen. Was jedoch im Zuge dieses Systems verloren geht, ist die Assoziierung von Objekten und weiterführenden Aktionen sowie die Verbindung zwischen mehreren Objekten. Programm 9.1 definiert eine Aktion, die es ermöglicht, eine Datei von einem Verzeichnis in ein anderes zu verschieben. Dabei wird festgelegt, dass die Objekte *von* und *nach* vom Typ Verzeichnis sind, sich voneinander unterscheiden und die Datei d sich in Verzeichnis *von* befindet. In der herkömmlichen Graphenplanung kann aufgrund dieser Aktionsbeschreibung berechnet werden, dass sich nach Ausführen der Aktion die Datei d im Verzeichnis *nach* befindet. Dieses Prinzip ist mit Annotationen alleine jedoch nicht abbildbar. Programm 9.2 versucht diesen Task zu implementieren, wofür zwei Beans benötigt werden, nämlich die Datei selbst und das Verzeichnis, in welches die Datei verschoben werden soll. Da immer nur ein Bean derselben Klasse pro Annotation möglich ist, kann auch nicht definiert werden, dass das Verzeichnis benötigt wird, in dem sich die Datei derzeitig befindet. Der kritische Punkt, der mit diesem Beispiel aufgezeigt werden soll, ist jedoch ein anderer. Es gibt keine Möglichkeit zu beschreiben, dass sich die Datei nach Ausführen des Tasks in dem übergebenen Verzeichnis befindet. Das wird zum Problem, wenn komplexere Szenarien damit umgesetzt werden sollen, vor allem, wenn mit genau diesem Verzeichnis weitergearbeitet werden soll, aufgrund der Information, dass sich die Datei dort befindet.

Programm 9.1: Definitionsbereich zum Verschieben einer Datei von einem Verzeichnis in ein anderes.

```
AKTION(Verschieben, {
  Parameter: (
    Verzeichnis(von) ∧ Verzeichnis(nach) ∧ Datei(d)
  ),
  Voraussetzungen: (
    NichtGleich(von, nach) ∧ BefindetSichIn(d, von)
  ),
  Effekte: (
    ¬BefindetSichIn(d, von) ∧ BefindetSichIn(d, nach)
  )
});
```

Programm 9.2: Umsetzung des Tasks zum Verschieben einer Datei. Es kann nur angegeben werden, dass eine Datei und ein Verzeichnis dafür benötigt werden.

```
@Preconditions({
  Datei.class,
  Verzeichnis.class
})
public class VerschiebeDatei extends AbstractTask {}
```

9.1.1 Aktionen textuell beschreiben

Viele *GraphPlan*-Implementierungen für Java und andere Programmiersprachen basieren auf textuell beschriebenen Aktionen ähnlich den Beispielen, die in dieser Arbeit angeführt wurden. Problembeschreibungen sind dabei laut dem originalen Algorithmus Konjunktionen von Wahrheitswerten. Das hat den Vorteil, dass Assoziationen zwischen Objekten und Aktionen, wie im vorherigen Abschnitt erwähnt, nicht verloren gehen. Dieses Konzept kann sehr einfach auf den Prototypen angewendet werden, da lediglich die Annotationen dahingehend geändert werden müssten, um mit textuellen Beschreibungen anstatt mit Beans zu arbeiten. Das restliche System müsste kaum adaptiert werden. Es gibt jedoch Szenarien, die auftreten können und berücksichtigt werden müssen. Das bezieht sich hauptsächlich auf das Erstellen von Problembeschreibungen, da mithilfe der selben textuellen Beschreibung gearbeitet werden muss. Das macht die Schnittstelle nicht ganz so einfach zu verwenden, wie bisher. Programm 9.3 zeigt eine Taskdefinition anhand von Annotationen, die auf textuellen Beschreibungen basieren, dafür wird eine neue Annotation `@Params` benötigt, um Objekte mit ihren Typen zu verknüpfen, was zuvor über die Beans gelöst wurde. In Programm 9.4 wird aufgezeigt, wie die Schnittstelle zum Scheduler aussehen könnte, wenn diese Änderungen ins System übernommen werden würden.

Programm 9.3: Geänderte Annotationen, die mit textuellen Beschreibungen arbeiten

```
1  @Params({
2    "Datei(d)",
3    "Verzeichnis(von)",
4    "Verzeichnis(nach)"
5  })
6  @Preconditions({
7    "NichtGleich(nach, von)",
8    "BefindetSichIn(d, von)"
9  })
10 @Effects({"BefindetSichIn(d, nach)"})
11 @NegativeEffects({"BefindetSichIn(d, von)"})
12 public class VerschiebeDatei extends AbstractTask {}
```

Programm 9.4: Erstellen einer Problembeschreibung anhand von textuellen Beschreibungen.

```
1  TaskState state = new TaskState();
2  state
3    .has(new StateValue("Datei", "d", datei))
4    .has(new StateValue("Verzeichnis", "von", von))
5    .has(new StateValue("Verzeichnis", "nach", nach))
6    .has(new StateRelation("NichtGleich", "von", "nach"))
7    .has(new StateRelation("BefindetSichIn", "d", "von"))
8    .wants(new StateRelation("BefindetSichIn", "d", "nach"));
9
10 scheduler.schedule(state);
```

9.2 Kommunikationsparadigmen

RMI wurde als Kommunikationsparadigma eingeführt, aufgrund der Einfachheit und der geringen Fehleranfälligkeit. Im Zuge von Überarbeitungsarbeiten am Prototypen wurde das System an das Eventloop Pattern angepasst. Dadurch wurde RMI als Paradigma zu einem größerem Hindernis als einem unterstützendem Tool, da pro RMI-Verbindung ein neuer Thread geöffnet wird. Außerdem sind viele Verfahren aus der Fehlertoleranz mit RMI nicht ausreichend implementierbar, wie zum Beispiel das Senden von Herzschlägen, um Ausfälle zu erkennen, da die Fehler in der Kommunikation stark abstrahiert werden. Vor der Weiterentwicklung des Systems sollte aus diesen Gründen RMI durch herkömmliche Socketkommunikation ersetzt werden. Das lässt sich im Zuge der Überarbeitungen auch mit *RxJava* verknüpfen, um ein großes Ekosystem an asynchronen Adaptern und anderen event-basierten Paradigmen ins System aufzunehmen. Durch diese Änderungen würde sich die Anzahl der Nachrichten, die ein einziger Knoten im System verarbeiten kann, stark erhöhen, um im Weiteren mit anderen Frameworks wie AKKA konkurrieren zu können.[1]

[1] Mit AKKA kann ein einziger Knoten, abhängig von dessen Leistung, bis zu 50 Millionen Nachrichten pro Sekunde verarbeiten und 2.5 Millionen Aktoren pro GB Heap anlegen.

Kapitel 10

Zusammenfassung

Das Problem der Jobgrößen in verteilen Lastenverteilungssystem ist kein triviales. Wie in dieser Arbeit ersichtlich wird, bilden sie einen wichtigen Teil bei der effektiven Verteilung von Lasten bezüglich CPU-Auslastung und Ähnlichem. Herkömmliche Strategien, wie das manuelle Verknüpfen von Tasks mittels Triggern wie in *Quartz*, lösen dieses Problem, steigern aber mit zunehmender Größe der Applikation die Komplexität. Der vorgestellte Ansatz dieser Arbeit, basierend auf der Graphen-Planung hingegen, bleibt, ähnlich dem Aktorensystem, in seiner Komplexität recht konstant, benötigt aber ein tiefes Verständnis des Algorithmus, der im Hintergrund verwendet wird. Die Verwendung der Graphenplanung bietet zwar die Möglichkeit, effizientere Pläne automatisiert zu berechnen, ist aber im Vergleich zum Aktorensystem in komplexeren Beispielen nicht so einfach zu verwenden bzw. anzuwenden. Der große Vorteil von Aktoren liegt darin, dynamisch andere Aktoren anlegen zu können. Ein Graphenplan ist nach der Berechnung statisch und wird genau in diesem Format abgearbeitet. Ein Aktor kann zur Laufzeit noch entscheiden, ob das Aufspalten in kleinere Teile überhaupt von Vorteil ist oder nicht. Das kann interessant sein, wenn zum Beispiel Webseiten wie bei einem Webcrawler verteilt analysiert werden sollen. Ein Aktor kann in diesem Fall pro Hyperlink einfach einen neuen Aktor erstellen, der diesen Link verfolgt und ebenfalls analysiert. In der Graphenplanung lässt sich ein solches System nur sehr schwer umsetzen. Die Probleme dieses Ansatzes, welche in Kapitel 9 aufgezeigt wurden, müssen für den Einsatz in einer marktreifen Applikation zuvor aber behoben werden.

Interessant waren während des Entstehungsprozesses dieser Arbeit meine Erfahrungen in Bezug auf die Konzepte verteilter Systeme, die ich von der We Are Developers Konferenz 2017 mitnehmen konnte. Zum einen wurde *RxJava* und welche Vorteile eine event-basierte Architektur haben kann im Zuge von vielen Projekten genauer erläutert. MobFox[1] wurde zum Beispiel auf dieser Architektur aufgebaut, um Millionen Anfragen in der Sekunde verteilt abarbeiten zu können. Auch Uber[2] setzt in diesem Zusammenhang sehr stark auf Node.js und das Eventloop Pattern. Zum anderen stellte *Palantir* interessante Ansätze für die Fehlertoleranz vor, sowie Sicherheitsmaßnahmen für verteile Suchsysteme. Für die Umsetzung dieser Ansätze und Paradigmen innerhalb des Prototypen war leider nicht genug Zeit, weswegen theoretisch auf diese Themen in

[1] http://www.mobfox.com/
[2] https://www.uber.com/

dieser Arbeit eingegangen wurde. Viele davon werden derzeit aber in die zweite Version des Prototypen eingebunden, damit dieser zukünftig den Namen Prototyp ablegen kann. Wichtig für mich als Entwickler waren vor allem die Konzepte verteilter Systeme, die während den Recherchen zusammengetragen wurden (Lastenverteilung, Auswahlalgorithmen, DHT, usw.). Bis dahin war mein Wissen bezogen auf verteilte Systeme begrenzt auf zwei Kurse an Universitäten. In vielen Aspekten der Informatik ist ein solches Wissen bereits ausreichend. Was mir aber schnell klar wurde ist, dass diese Annahme nicht auf verteilte Systeme zutrifft. Beim Lösen eines Problems in solchen Systemen öffnen sich immer wieder Türen zu weiteren Dingen, die berücksichtigt und behandelt werden müssen, um ein System zu schaffen, welches auf Grenzfälle eingehen und sich von möglichen Fehlerfällen selbst erholen kann.

Kapitel 11

Inhalt der CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

11.1 PDF-Dateien

Pfad: /

thesis.pdf	Masterarbeit
project-report.pdf . . .	Projektbericht

11.2 Online Quellen

Pfad: /online-literature

BitTorrentKademlia/bep_005.html [27]

ServiceLocator/The Service Locator Pattern.html [28]

11.3 Prototyp

Pfad: /prototype

README.txt	Instruktionen zum Ausführen des Prototypen
bin/commander-0.9.jar	Executable
src	Java Projekte und Source-Dateien

11.4 Sonstiges

Pfad: /images

*.eps	Originale Adobe Illustrator-Dateien

Quellenverzeichnis

Literatur

[1] Seema Balhara und Kavita Khanna. „Leader Election Algorithms in Distributed Systems“. In: *International Journal of Computer Science and Mobile Computing*. Bd. 3. 6. 2014, S. 374–379 (siehe S. 37, 39).

[2] Avrim L. Blum und Merrick L. Furst. „Fast Planning Through Planning Graph Analysis“. *Artificial Intelligence* 90 (1997), S. 281–300 (siehe S. 8, 12).

[3] Daniel Bryce und Subbarao Kambhampati. „A Tutorial on Planning Graph Based Reachability Heuristics“. *AI Magazine* 28 (2007), S. 47–83 (siehe S. 8).

[4] C. Chen, C. B. Yao und S. M. Liang. „Towards Practical Virtual Server-Based Load Balancing for Distributed Hash Tables“. In: *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. Sydney, NSW, Australia, Dez. 2008, S. 35–42 (siehe S. 48, 49).

[5] G. Chiola. „An Empirical Study of Data Redundancy for High Availability in Large Overlay Networks“. In: *Proceedings of the Second International Workshop on Hot Topics in Peer-to-Peer Systems*. San Diego, CA, USA, Juli 2005, S. 43–50 (siehe S. 48, 49).

[6] Rick Copeland. *MongoDB Applied Design Patterns*. O'Reilly, 2013 (siehe S. 34).

[7] Flavin Cristian. „Understanding Fault-Tolerant Distributed Systems“. *Communications of the ACM* 34.2 (Feb. 1991), S. 56–78 (siehe S. 35).

[8] Richard E. Fikes und Nils J. Nilsson. „STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving“. In: *Proceedings of the Second International Joint Conference on Artificial Intelligence*. IJCAI'71. London, England: Morgan Kaufmann Publishers Inc., 1971, S. 608–620 (siehe S. 9).

[9] Ian Foster und Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999 (siehe S. 2).

[10] Felix C. Freiling, Rachid Guerraoui und Petr Kuznetsov. „The Failure Detector Abstraction“. *ACM Computing Surveys* 43.2 (Feb. 2011), 9:1–9:40 (siehe S. 36).

[11] K. Garala, N. Goswami und P. D. Maheta. „A Performance Analysis of Load Balancing Algorithms in Cloud Environment“. In: *Proceedings of the 2015 International Conference on Computer Communication and Informatics (ICCCI)*. Jan. 2015, S. 1–6 (siehe S. 27, 30).

[12] J. W. Hanna und J. D. Johannes. „A Reliable Distributed System Using Dual Level Fault Tolerance“. In: *Proceedings of the Southeastcon '92*. Birmingham, AL, USA, USA, Apr. 1992, 610–613 vol.2 (siehe S. 36).

[13] Thomas A. Henzinger u. a. „Scheduling Large Jobs by Abstraction Refinement“. In: *Proceedings of the Sixth Conference on Computer Systems*. New York, NY, USA, 2011, S. 329–342 (siehe S. 5).

[14] Andreas Kuehlmann, Kenneth L. McMillan und Robert K. Brayton. „Probabilistic State Space Search“. In: *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '99. San Jose, California, USA: IEEE Press, 1999, S. 574–579 (siehe S. 8).

[15] Monika Kushwaha und Saurabh Gupta. „Various Schemes of Load Balancing in Distributed Systems - A Review“. *International Journal of Scientific Research Engineering and Technology* 4.7 (2015), S. 741–7487 (siehe S. 27–29).

[16] F. Liu und J. B. Weissman. „Elastic Job Bundling: An Adaptive Resource Request Strategy for Large-Scale Parallel Applications“. In: *Proceedings of the SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2015, S. 1–12 (siehe S. 5).

[17] Gurmeet Manku, Mayank Bawa und Prabhakar Raghavan. *Symphony: Distributed Hashing in a Small World*. Technical Report 2003-79. Stanford InfoLab, März 2003 (siehe S. 46).

[18] Petar Maymounkov und David Mazières. „Kademlia: A Peer-to-Peer Information System Based on the XOR Metric“. In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, S. 53–65 (siehe S. 47).

[19] Michael David Mitzenmacher. „The Power of Two Choices in Randomized Load Balancing“. PhD dissertation. University of California, Berkeley, 1996 (siehe S. 48).

[20] Tao Qian, F. Mueller und Yufeng Xin. „A Real-Time Distributed Hash Table“. In: *Proceedings of the 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2014, S. 1–10 (siehe S. 43, 44, 48).

[21] M. Randles, D. Lamb und A. Taleb-Bendiab. „A Comparative Study into Distributed Load Balancing Algorithms for Cloud Computing“. In: *Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*. Apr. 2010, S. 551–556 (siehe S. 30).

[22] Simon Rieche, Leo Petrak und Klaus Wehrle. „Comparison of Load Balancing Algorithms for Structured Peer-to-Peer Systems“. In: *In Proceedings of the Workshop on Algorithms and Protocols for Efficient Peer-to-Peer Applications 2004*. 2004, S. 214–218 (siehe S. 48, 49).

[23] William Stallings, Hrsg. *Betriebssysteme, Prinzipien und Umsetzung*. 4. Aufl. München, Germany: Pearson Studium, 2003 (siehe S. 20).

[24] Andrew S. Tanenbaum und Maarten van Steen, Hrsg. *Verteile Systeme, Prinzipien und Paradigmen*. 2. Aufl. München, Germany: Pearson Studium, 2008 (siehe S. 2, 23, 34–37, 43–46).

[25] S. Tilkov und S. Vinoski. „Node.js: Using JavaScript to Build High-Performance Network Programs“. *IEEE Internet Computing* 14.6 (Nov. 2010), S. 80–83 (siehe S. 24).

[26] Chee Shin Yeo u. a. „Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers“. In: *Handbook of Nature-Inspired and Innovative Computing*. Springer US, 2006. Kap. 2, S. 521–551 (siehe S. 2).

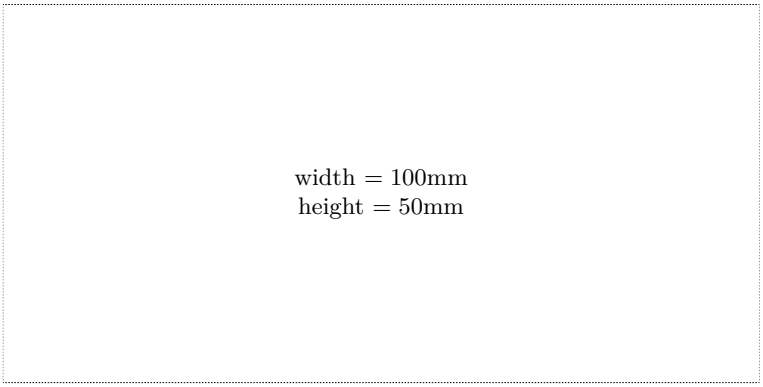
Online-Quellen

[27] BitTorrent. *BitTorrent DHT Protocol*. URL: http://www.bittorrent.org/beps/bep_0005.html (besucht am 27. 08. 2017) (siehe S. 47, 55).

[28] Microsoft. *The Service Locator Pattern*. URL: https://msdn.microsoft.com/en-us/library/ff648968.aspx (besucht am 21. 07. 2017) (siehe S. 21, 55).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —