Bachelorarbeit

Leichtgewichtige Javascript Middleware für verteilte Prozessausführung

Michael Glatzhofer

Institute of ComputerGraphics and KnowledgeVisualisation, TU Graz

www.cgv.tugraz.at

Univ.-Prof. Dr.rer.nat. M.Sc. Tobias Schreck, Dipl.-Inf. Robert Gregor

9. August 2016

Abstract

Häufig ist es notwendig, Algorithmen auf eine große Anzahl von Input-Dateien anzuwenden. Dabei ist es bei einer verteilten Ausführung nicht notwendig den Algorithmus zu parallelisieren. Es ist ausreichend, den gegebenen Algorithmus auf mehreren Geräten mit unterschiedlichen Inputs zu starten. Existierende Tools wie MapReduce oder Apache Spark sind dafür ausgelegt Algorithmen zu parallelisieren, und dadurch komplexer als notwendig. Diese Arbeit beschäftigt sich mit dem Design, der Implementierung und Evaluierung einer leichtgewichtigen JavaScript Middleware. Die Eigenschaft "leichtgewichtig" bezieht sich auf den Sourcecode- und API-Umfang. Tasks mit langen Laufzeiten erfordern aus Sicht der Usability die Möglichkeit abgebrochen zu werden, eine Progress-Anzeige und Informationen über den Endzustand des ausgeführten Algorithmus.

Diese Arbeit analysiert zunächst bestehende und geeignete Konzepte und Technologien für eine solche Middleware und stellt danach eine Implementierung einer Middleware mit JavaScript, Node.js, Websockets und asynchronem API vor. Da durchgehend ein asynchrones API verwendet wird, benötigen die auf der Middleware aufgebauten Anwendungen keine Synchronisierungsmechanismen wie zum Beispiel Semaphoren. Die Skalierbarkeit dieses Prototypen wird in Kapitel 6 anhand eines Algorithmus zum Vorverarbeiten von 3D Meshes und einer verteilten Primzahlen-Suche analysiert und zeigt lineare Skalierbarkeit bis zu 16 Worker.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die verteilte Ausführung von Algorithmen ist ein häufiger Lösungsansatz zur Minimierung ihrer Laufzeit. Systeme wie MapReduce und Apache Spark haben in den letzten Jahren erheblich an Verwendung zugenommen. Diese Systeme reduzieren den Deployment-Aufwand auf die Installation eines Interpreters auf den ausführenen Geräten und bieten dem Anwender dadurch eine stark vereinfachte Handhabung. Denoch sind es Systeme die zur Verwendung sehr viel Wissen über die Technologie voraussetzen.

Muss ein Algorithmus auf viele Input-Dateien angewendet werden, ist es nicht notwendig den Algorithmus selbst zu parallelisieren. Eine Ausführung auf mehreren Rechnern mit unterschiedlichen Input-Dateien ist ausreichend. Für diesen Fall könnte ein System, das weniger Einarbeitungszeit als Apache Spark benötigt, gefunden werden. Shell Scripts können diese Aufgabe übernehmen, allerdings nicht in heterogenen Netzwerken. Remote Procedure Calls bieten ein einfaches API, sind aber nicht für Tasks mit langen Ausführungszeiten optimiert und von komplexen Deployment Schritten begleitet. Bei Tasks mit langen Ausführungszeiten ist eine Progress-Anzeige und die Möglichkeit die Ausführung zu stoppen im praktischen Gebrauch notwendig, da es ansonsten zur Verschwendung von Rechner-Ressourcen und Wartezeiten kommt. Auch aus Usability Gründen sind diese Features wünschenswert.

Ziel dieser Arbeit ist es, eine für oben genannte Aufgaben optimierte leichtgewichtige Middleware zu schaffen, die eine möglichst große Bandbreite an Geräten und Betriebssystemen abdeckt und ein einfaches Scripting API bietet. Kapitel 2 zeigt die Ergebnisse einer Analyse verwandter Technologien und Themenbereiche. Die für den Prototyp ausgewählten Technologien und Konzepte werden in Kapitel 3 aufgelistet und begründet.

Kapitel 4 und 5 dokumentieren das Design des Protokolls und des Prototypen. Dabei werden grundlegende Probleme und Lösungen der verteilten Programmierung, welche für diese Implementierung relevant sind, näher betrachtet. Mit der Absicht auch andere Algorithmen mit diesem Tool verteilt auszuführen wurde ein Scripting API eingeführt, sodass der Prototyp in zwei Komponenten unterteilt werden kann: in eine Middleware und dem Script das den gegebenen Algorithmus ausführt. Im Zuge der Evaluierung wurden mehrere Scripts geschrieben, wie zum Beispiel eine verteilte Primzahlen-Suche und ein Script welches einen Algorithmus zum Vorverarbeiten von 3D Meshes auf mehrere Input-Dateien anwendet. Die auf den Workern ausgeführten Algorithmen sind in C++ implementiert. Der entwickelte Prototyp verwendet JavaScript, Node.js, Websockets, und den Active Object Pattern remote Prozesse zu dirigieren. Progress-Informationen werden von den C++ Prozessen an die Middleware über Betriebssystem-Pipes übergeben und von dieser an das auftraggebende Gerät weitergereicht. Die Input-Dateien und C++ Binarys werden nicht von der Middleware verwaltet, und müssen deshalb auf allen Rechnern zur Verfügung stehen. Derzeit wird dafür ein verteiltes Dateisystem verwendet.

Eine Diskussion der Vor- und Nachteile des Designs erfolgt in Kapitel 7. Da eine Bewertung der Simplizität des Scripting APIs formal nicht trivial ist, liegt der Schwerpunkt der Diskussion auf Skalierbarkeit. Das Konzept ist auf einer niedrigeren Abstraktionsebene angesiedelt und lässt die Implementierung des Schedulers offen. Desshalb werden in diesem Kapitel auch Eigenschaften des Systems und daraus resultierende Anforderungen an erweiterte Scheduler-Implementierungen gezeigt. Anhand des Algorithmus zum Vorverarbeiten von 3D Meshes wird in Kapitel 6 gezeigt, dass nur ein minimaler Overhead benötigt wird, sofern die Anzahl der mit einem Server verbundenen Workern korrekt dimensioniert wird. Bei den genannten Beispielen wird lineare Skalierbarkeit mit bis zu 16 Worker gezeigt. Die JavaScript Implementierung kann im worst Case, wenn die Worker Laufzeit minimal ist, bis zu 4 Worker verwenden. Um mehr Worker zu unterstützen, ist eine Verbesserung des Network Overlays notwendig.

Kapitel 2

Ähnliche Arbeiten und Technologien

2.1 Konzepte

2.1.1 Message-Oriented Middleware vs Remote Procedure Calls

Middleware Systeme können nach mehreren Gesichtspunkten klassifiziert werden. Im Folgenden wird die Klassifizierung nach [BK03] verwendet und auf die Klassen "Procedure Oriented", "Object Oriented", und "Message-Oriented Middleware" eingegangen, da sie unter Berücksichtigung der Requirements mögliche Lösungsansätze sind.

"Procedure Oriented" und "Object Oriented" Systeme werden meißt unter dem Begriff Remote Procedure Call (RPC) zusammengefasst, und sind häufig als Request-Response Protokolle implementiert, wie zum Beispiel Java RPC oder CORBA. Einen RPC abzubrechen ist in diesem Kontext nur möglich, indem man die Ausführung an die TCP Connection bindet, das hieße die Ausführung abbzubrechen, falls die TCP Connection geschlossen wird. Eine weitere Message für den Abbruch zu spezifizieren würde es ermöglichen, TCP Connections aufrecht zu erhalten wenn ein RPC abgebrochen wird. Dies widerspricht aber dem Request-Response Konzept. RPC Implementierungen wie [1&16] setzen dieses Konzept um. Bei der Evaluierung von RPC Technologien konnte keine gefunden werden, die den Fortschritt der Ausführung am Client zur Verfügung stellt. Um dies mit RPC zu realisieren, muss es im Application Layer implementiert werden.

"Message-Oriented Middleware" Systeme (MOM) bieten mehr Freiheit in der Hinsicht, dass Message-Sequenzen beliebig modelliert werden können. Ein Remote Call mit Progress und Cancel Funktionalität kann relativ simpel mit vier Messages modelliert werden: Request, Progress, Cancel und Response. Bei dieser Vorgehensweise muss jede der erwähnten Messages eine ID enthalten, welche auf den Aufruf verweist.

Beide Middleware Konzepte benötigen für die Progress und Cancel Funkionalität Implementierungen im Application Layer - eine Aufgabe, die dem Anwender abgenommen werden könnte.

2.1.2 Asynchrone API

MOM Systeme verwenden meist asynchrone APIs, während RPC Systeme meist synchrone APIs verwenden. Es ist aber auch möglich, asynchrone APIs für RPC zu verwenden wie Falkner [FCO99] gezeigt hat.

Asynchrone APIs, die mit Callbacks arbeiten, führen zu schwer lesbaren Code (pyramid of doom). Um dies zu vermeiden können Futures und/oder Promises verwendet werden [BJH77]. Promises sind Objekte, die das Ergebnis eines Funktionsaufrufs repräsentieren. Bei den meisten Programmiersprachen kann dies neben dem Returnwert auch eine Exception sein. Verwendet man Promises, wird das Ergebnis nicht durch Schlüsselwörter wie return oder throw definiert, sondern an das Promise Objekt übergeben. So ist es möglich, auch innerhalb von Callbacks oder nach dem Verlassen der Funktion ein Ergebnis zu definieren.

Futures ergänzen dieses Konzept an der aufrufenden Stelle. Asynchrone Funktionen geben beim Aufruf ein Future Objekt zurück. Dieses kann verwendet werden um auf die Terminierung zu warten, beziehungsweise einen Eventhandler zu registrieren. Wenn die asynchrone Funktion das Ergebnis an die Promise übergibt, gibt es diese an die Future weiter, welche in den Zustand Terminated übergeht. Ist die Future in diesem Zustand, kann der Returnwert von ihr abgelesen werden [BJH77].

2.1.3 Active Object Pattern aka Actor Pattern

Bei der Konzeptionierung von Serversystemen muss ein Threading Model gewählt werden. Der Einsatz von multithreaded Modellen für ein Scripting API ist zu vermeiden, da es hohe Anforderungen an den Anwender im Bereich der Synchronisierung stellt.

Eine Möglichkeit Semaphoren zu vermeiden ist der Active Object Pattern [SSRB13]. Dieser verbindet eine Queue, in der Actions liegen, mit einem Thread welcher die Actions abarbeitet. Es muss lediglich das Einfügen und Entfernen aus der Queue als atomare Operation ausgeführt werden.

2.2 Technologien

2.2.1 Cluster- Grid- und Cloudcomputing

Cluster- Grid- und Cloudcomputing haben eine Gemeinsamkeit: Sie dienen der Nutzung von verteilten Ressourcen. Ihre Unterschiede und die Abgrenzung zueinander sind nicht einheitlich definiert. Im Folgenden wird die Klassifizierung nach [SK11] verwendet und ein kurzer Überblick in Bezug auf örtliche Ausdehnung, Ressourcen Allocation, Taskgröße, Heterogenität, Skalierbarkeit und Task Deployment gegeben.

Cluster Computing Systeme sind homogene, für lokale Netzwerke mit hohem Durchsatz optimierte Systeme. Die Ressourcenverteilung wird von zentraler Stelle aus organisiert und limitiert damit die Skalierbarkeit. Die Taskgröße ist nicht limitiert, dadurch eignen sie sich für verteilte Batch Verarbeitung. Häufig wird MPI, eine Message Passing Middleware, verwendet. Das Deployment der damit implementierten Programme ist nicht standardisiert und verlangt vom User detailliertes Wissen über den Cluster.

Grid Computing Systeme sind heterogene, für Netzwerke mit globaler Ausdehnung und geringem Durchsatz optimierte Systeme [FK03, Fos06]. Typischerweise werden dabei mehrere lokale Netzwerke zu einem Grid verbunden, wodurch eine hierarchische Struktur entsteht. Die Ressourcen Verteilung folgt ebenfalls dieser hierarchische Struktur, wodurch eine hohe Skalierbarkeit gegeben ist. Die Taskgrößen sind nicht limitiert, dadurch eignen sie sich für verteilte Batch Verarbeitung. Es existieren Erweiterungen wie MPICH-G2, die automatisches Job Deployment unterstützen.

Cloud Computing Systeme sind heterogene Systeme mit globaler Ausdehnung. Sie sind für kleine Taskgröße designed und skalieren in Bezug auf die Anzahl dieser Tasks, aber nicht in Bezug auf die Große eines Tasks. Diese Eigenschaft wird durch die Platform as a Service Eigenschaft erreicht, die das Task Deployment soweit vereinfacht, dass es on demand und automatisch geschehen kann.

2.2.2 Apache Spark

Apache Spark ist ein Cluster Computing System, das auf einer MOM aufbaut. Ein zu verteilender Algorithmus kann innerhalb eines Scripts implementiert werden. Apache Spark API basiert auf Resilient Distributed Datasets (RDD). RDDs bieten Methoden wie map, reduce, groupBy, filter und viele mehr.

Bei dem Aufruf dieser Funktionen unterteilt Apache Spark das RDD in Partitionen, wobei jeweils eine Partition von einem Worker verarbeitet wird. Die hier als Beispiele genannten Methoden sind Funktionen höherer Ordnung, machen also klar, das auch ausführbarer Code an die Worker übergeben werden muss, damit dieser seine Partition abarbeiten kann. Dies kann als versteckter Deployment Schritt gesehen werden und ist auch mit ein Grund für die einfache Handhabung von Apache Spark.

Darüber hinaus wird der User von technischen Details wie Fehlerkorrekturen und dem Versenden von Netzwerk Messages entbunden. Apache Spark unterstützt mehrere Persistenz Technologien, darunter HDFS, Hadoop und Cassandra [ZCD*12].

Kapitel 3

Anforderungen und verwendete Technologien

In diesem Kapitel werden zunächst die Anforderungen an die Middleware zusammengefasst und analysiert. Danach die verwendeten Technologien und Konzepte aufgelistet und begründet, warum die Entscheidung auf sie gefallen ist.

3.1 Anforderungen

- Es soll eine Middleware geschaffen werden, die auf einem möglichst großen Spektrum von Betriebssystemen und Geräten lauffähig ist.
- Application Layer sollten als Scripts realisiert werden, die von der Middleware ausgeführt werden. Diese Scripts werden im folgenden JobScript genannt.
- Der Deployment Aufwand soll minimiert werden. Wie bei Apache Spark sollte der User nur ein Script schreiben müssen, welches das Ausführungsverhalten des gesamten Systems definiert, also auch den auf anderen Geräten ausgeführten Code enthält. Die Teile des Scripts welche auf anderen Geräten ausgeführt werden, sollten on demand auf diese verteilt werden.
- JobScripts sollten frei von Synchronisierungsmechanismen wie Semaphoren sein, da ansonsten zu viel technisches Know-how für die Implementierung von JobScripts notwendig ist.
- JobsScripts sollen es ermöglichen bei Funktionsaufrufen ein Gerät zu spezifizieren, auf dem die Funktion ausgeführt wird. Da bei RemoteJobs das Gerät explizit angegeben werden muss, muss innerhalb der Scripting Sandbox zumindest Zugriff auf die verbundenen, oder alle im Netzwerk zur Verfügung stehenden Geräte gegeben sein. Somit kann der Anwender eigene Scheduling Strategien implementieren. Pro Rechner muss CPU Auslastung, verfügbarer Speicher und verfügbare Interpreter abgelesen werden können. Im Folgenden wird diese Datenstruktur NetInfo genannt, siehe 5.2.3.4.
- Argumente und Returnwert oder Exceptions sollen von der Middleware transportiert werden.
- Auch das Starten von Remote Prozessen sollte auf dieses Art möglich sein. Solche Remote Calls werden im folgenden RemoteJobs genannt. Es wird angenommen, dass RemoteJobs mit einer sehr langen Ausführungszeit zum Einsatz kommen, deshalb müssen diese abbrechbar sein. Außerdem muss es möglich sein, den Progress von RemoteJobs an der auftraggebenden Stelle abzulesen.

3.2 Verwendete Technologien

3.2.1 ECMA Script aka JavaScript

Ein JobScript enthält den Code aller beteiligten Geräte. Funktion hohrere Ordnung enthalten Funktionen in der Argumentliste. Bei Remote Aufrufen müssen die Argumente an die Gegenstelle gesendet werden, bei Porgrammiersprachen wie C++ ist es schwierig eine Serialisierbare Repräsentation einer Funktion zu erhalten. Die Implementierung eines eigenen Compilers oder Präprozessors soll umgangen werden. Programmiersprachen mit Reflection würden dies ermöglichen, allerdings mit erheblichem Aufwand. JavaScript macht es sehr einfach Stringrepräsentationen einer Funktion zu erhalten.

Javascript Interpreter sind auf nahezu allen Betriebssystemen vorhanden. Auch mobile Endgeräte sind meist mit Browsern ausgestattet, die Javascript unterstützen. Dadurch ist JavaScript eine der am häufigsten zur Verfügung stehenden Laufzeitumgebungen. JavaScript unterstützt kein Multithreading - auf Servern werden Prozesse anstatt Threads verwendet [Nod16]. Webworker unterstützen kein Shared Memory, und werden im folgenden nicht als Threads betrachtet.

3.2.2 Websockets

Für die Synchronisierung der NetInfo Datenstruktur muss bei Änderungen der Geräteeigenschaften eine Message an den Server und von diesem weiter an die Clients gesendet werden. Vor allem das Weiterleiten an die Clients kann mit Websockets effizienter, das heißt ohne Polling, realisiert werden. RemoteJobs müssen dem auftraggebenden Gerät Progress Updates zukommen lassen. Auch hier kann Polling durch die Verwendung von Websockets vermieden werden.

3.2.3 Node.js und Active Object Pattern

JavaScript in Browsern und Node.js verwenden ein Threading Model, das einem Active Object Pattern entspricht, wobei ein ganzer Node.js Prozess sowie ein Web Worker als Active Object Pattern gesehen werden kann. Darauf aufbauend kann eine MOM sehr einfach implementiert werden. Um die Terminierung, das Abbrechen und Timeouts von RemoteJobs zu realisieren, muss eine State Machine auf Client- sowie Serverseite implementiert werden (siehe Kapitel 4). Die Transitions dieser State Machine werden durch Netzwerk Messages, User Inputs und Timer Events ausgelöst.

Da Webbrowser und Node.js kein Multithreading mit Shared Memory unterstützen, ist ein Asynchronous API unumgänglich. Ein synchrones API würde entweder eine hohe Anzahl von Prozessen oder Web Worker benötigen, beziehungsweise starke Latenz Einbußen mit sich bringen.

Kapitel 4

Protokoll Design

Das hier beschriebene Protokoll erweitert das RPC Konzept. Aufrufe können abgebrochen werden, und der Progress kann am Client verfolgt werden, ohne zusätzliche Remote Aufrufe zu starten. Das Protokoll wird in einem hierarchischen Client-Server Overlay Network (HCSNO) verwendet. Da in einem solchen Netzwerk Jobs vom Client an den Server und von diesem an Worker gegeben werden, wird im Folgenden zwischen Auftraggeber und Auftragnehmer unterschieden, wobei der Server beide Rollen einnimmt.

Zustand	Gültige Transitions
Init	call
Running	update, cancel, returnFail, returnOk
Canceling	update, returnCanceled, returnFail, returnOk
Ok	-
Canceled	-
Failed	-

Tabelle 4.1: Zustände und Transitions eines Jobs. Siehe Abbildung 4.2 für das entsprechende FSM Diagramm.

Jeder teilnehmende Prozess besitzt eine global eindeutige ID, im Folgenden NodeID genannt. Jeder Job egal ob remote oder nicht, erhält ebenfalls eine ID, im Folgenden JobID genannt.

4.1 Network Messages

Alle Messages enthalten eine JobID und beziehen sich jeweils auf genau einen Job. Zudem enthält jeder Job die JobID des ParentJobs, also jene des Jobs, von dem er erstellt wurde. Alle Messages sind idempotent. Grundsätzlich entspricht eine Network Message einer State Machine Transition, nur die Return Message repräsentiert die returnOk, returnFail und returnCanceled Transitions. Die Return Message enthält ein zusätzliches Flag, das zur Unterscheidung dient.

4.1.1 Call Message

Auftraggeber → *Auftragnehmer*
Führt einen Algorithmus am Auftragnehmer aus. Der Algotithmus ist als String codierte JavaScript Funktion in der Message enthalten. Der Enthaltene Code hat seinen Ursprung im Application Layer des Auftraggebers, beim Erstellen des Jobs (siehe 5.3).

4.1.2 Cancel Message

Auftraggeber → *Auftragnehmer*
Nur der Auftraggeber kann die Ausführung mit dieser Nachricht abbrechen. Cancel Messages führen nicht unbedingt zum Endzustand Canceled. Falls der Auftragnehmer nicht im Zustand Running ist, wird die Cancel Message ignoriert.

4.1.3 Update Message

Auftragnehmer → *Auftraggeber*
Der Auftragnehmer kann mit dieser Message Zwischenberichte wie den Progress, für Menschen lesbare Statusinfos oder Daten an den Auftragnehmer senden. Im Gegensatz zur Return Message darf diese mehrfach gesendet werden.

4.1.4 Return Message

Auftragnehmer → *Auftraggeber*
Der Auftragnehmer signalisiert dass der Job erfolgreich ausgeführt wurde, oder eine Exception aufgetreten ist. Return Messages können die State Machine Transitions returnOk, returnFail oder returnCanceled auslösen. Ein Flag in der Message bestimmt die Transition.

4.2 Job State Machine

Ein grundlegendes Problem von Distributed Computing ist es, am Client den Zustand von Remote Calls zu verfolgen. Acknowledges können auf mehreren Layern zum Einsatz kommen, zum Beispiel nur am Tansport Layer. In diesem Fall wird meist davon ausgegangen, dass eine erfolgreiche Übertragung auch zu einer erfolgreichen Ausführung führt. Algorithmen, bei denen diese Annahme nicht getroffen werden kann, benötigen auch Acknowledges auf dem Application Layer. Abbildung 5.5 zeigt eine simple State Machine für einen RPC der nicht abgebrochen werden kann.

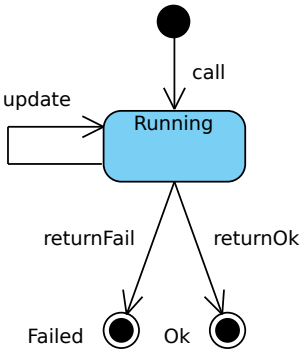

Abbildung 4.1: State Machine eines nicht abbrechbaren RPC

An dieser Stelle sollte beachted werden, dass auch diese State Machine verteilt ist. Sie stellt den Zustand des Remote Calls dar, dieser existiert real am Client und am Server. Oder mit anderen Worten: Dieser Zustand wird im Stub und im Skeleton gehalten. Entsprechende Netzwerk Messages synchronisieren den Zustand zwischen Client und Server. Die ReturnOk Transition emittiert eine positive Return Message, die ReturnFail Transition emittiert eine negative Return Message. Die Call Transition wird durch eine Call Message ausgelöst.

Auch für den Fall, dass kein Abbruch des RPC unterstützt wird und der Server nur am Ende der Ausführung eine Message mit dem Erfolgszustand an den Client sendet, ist es nicht immer möglich, diese State Machines zu synchronisieren. Siehe Zwei-Armeen-Problem [Tan]. Denn der Client könnte durch ein Timeout Event, das genau dann eintritt, wenn die Return Message vom Server auf dem Weg zum Client ist, in den Failed Zustand übergehen, während der Server bereits im Ok Zustand ist.

Wird das System so erweitert, dass Remote Calls abgebrochen werden können (siehe Abbildung 4.2), trifft man auf dasselbe Problem wenn vom Client eine Abbruch Meldung an den Server gesendet wird und der Server bereits eine Erfolgsmeldung gesendet hat, diese aber noch nicht am Client angekommen ist. Eine Strategie um dieses Problem zu minimieren ist es, den Client nicht in den Canceled State übergehen zu lassen ohne vom Server eine Bestätigung erhalten zu haben. Im Detail bedeutet das, dass der Client in den Canceling State übergeht, eine Nachricht an den Server sendet mit dem Auftrag die Ausführung abzubrechen und dann vom Server den Endzustand erhält. Soweit scheint es, dass beide am Ende im selben Zustand sind, jedoch muss auch der Canceling State mit einem Timeout überwacht werden, was wiederum zum Zwei-Armeen-Problem führt.

Will man nicht nur am Ende der Ausführung vom Server über dessen Zustand informiert werden, sondern auch während der Ausführung, zum Beispiel über den Progress oder Warnings, so können ebenfalls Race Conditions beim Abbruch auftreten, wenn nicht diese Art der Terminierung verwendet wird. Der Client könnte durch ein Timeout bereits im Failed State sein, während der Server noch Running States publiziert.

Dabei wird auch offensichtlich, dass Ressourcen verschwendet werden, während am Client bereits davon ausgegangen wird, dass die Ausführug fehlgeschlagen ist, der Server aber noch daran arbeitet. In diesem Fall können Ergebnisse nicht vom Client angenommen werden, weil am Client die Ausführung bereits als Fehlerhaft deklariert wurde.

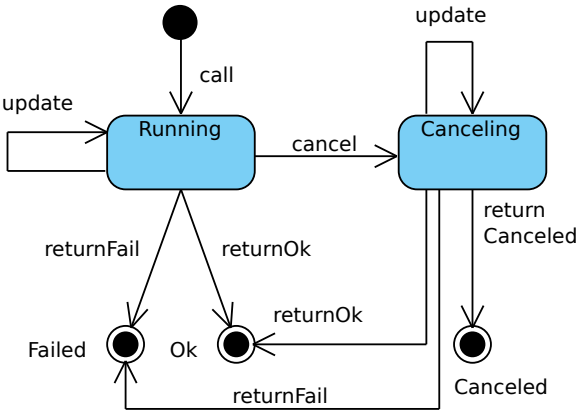

Abbildung 4.2: Erweiterte State Machine eines abbrechbaren RPC

Die Verschwendung dieser Ressourcen kann minimiert werden, in dem der Client im Fall eines Timeouts den Remote Call am Server durch eine Cancel Message beendet - minimiert deshalb, weil immer noch Netzwerkfehler einen solchen Abbruch verhindern können. Im Folgenden werden Remote Calls, welche diese State Machine implementieren, Jobs genannt. Jobs können aber auch rein lokal für asynchrone Aufrufe verwendet werden. Das handling von verlorenen Messages ist in diesem Fall zwar nicht notwendig, dennoch können sie hilfreich sein beim auffinden von Implmentierungsfehlern und Race Conditions (siehe Kapitel 5).

4.3 Job Trees und Call Graphen

In einem HCSNO ist es üblich Aufgaben zu delegieren. Der Client gibt Aufträge an den Server, der Server an die Worker. Die einzelnen durchlaufenen Code Stücke können als Directed Acyclic Graph (DAG), der Ahnlichkeiten zu einem lokalen Call Graphen hat, dargestellt werden [YB05]. Dieser DAG beschreibt den durchlaufenen Workflow über alle Rechner. In Peer-to-peer (P2P) Netzwerken können diese Graphen noch größere Tiefen erreichen. Jedem Knoten im DAG ist eine Workflow Logic zugeordnet. Typische Workflows sind:

- Worker werden parallel gestartet und müssen alle erfolgreich terminieren
- Worker werden parallel gestartet, einer muss erfolgreich sein (Redundanz)
- Worker werden konsekutiv gestartet, alle müssen erfolgreich sein
- Pooling. N Worker arbeiten parallel, jedoch gibt es mehr Aufträge als Worker. Schließt ein Worker einen Job ab, beginnt er mit einem der übrigen Jobs, bis alle abgearbeitet wurden

Die Vollständigkeit dieser Liste ist nicht gegeben. Der Anwender muss also die Möglichkeit haben, diese Liste zu erweitern. Die Javascript Bibliothek async [Asy16] verwendet ein ähnliches Konzept für asynchrone Funktionsaufrufe, und zeigt eine beeindruckende Liste an möglichen Workflows.

Der Workflow DAG dieser Implementierung wird im folgenden Job Tree genannt. Jeder Knoten im Job Tree ist eine Job State Machine. Delegiert ein Job A eine Aufgabe an einen anderen Job B, wird A im folgenden ParentJob, und B SubJob genannt.

Hat ein Knoten in einem Call Graphen mehrere Kinder, bedeutet dies immer sie wurden Konsekutiv ausgeführt. Im Kontext dieser Arbeit muss es möglich sein Unteraufgaben parallel auszuführen, genauer betrachtet muss es sogar möglich sein den Startzeitpunkt für jede Unteraufgabe frei festzulegen (siehe Pool Workflow). Die Implementierung des Prototypen verwendet einen Strategy Pattern [GHJV05] um den Anwender der Middleware die Möglichkeit zu geben eigene Startsequenzen zu definieren.

Der Zustand des ParentJobs ist von den Zuständen der SubJobs, und der ParentJob Terminate Strategy abhängig. Die angestrebte Vereinfachung des Middleware APIs beruht im Wesentlichen darauf, dass die Implementierung des Workflows nicht im Application Layer statt findet, sondern das eine SubJob Start Strategy und eine ParentJob Terminate Strategy aus einer Menge von vordefinierten Strategien ausgewählt wird.

4.4 Hierarchische Job State Machine

Soll ein Job, zum Beispiel am Server, nur die geeigneten Worker ermitteln und die ihm übergebene Aufgabe an diese delegieren, kann die Funktion Job.delegate verwendet werden um dem Job SubJobs hinzuzufügen. Job.delegate erwartet eine ParentJob Terminate Strategy, eine SubJob Start Strategy, sowie eine Liste der Subjobs. Anstatt der Liste der SubJobs kann auch eine Factoryfunktion verwendet werden. Job.delegate bindet die State Machine des ParentJobs and die der SubJobs, es entsteht eine hierarchische FSM (siehe Abbildung 4.3).

Der Aufruf von Job.delegate führt die SubJob Start Strategy aus. Diese kann nun entscheiden welche SubJobs gleich zu Beginn gestartet werden sollen. Terminiert ein SubJob wird die ParentJob Terminate Strategy aufgerufen. Diese muss zunächst entscheiden ob der Parent terminieren soll, wenn nicht kann sie bei bedarf weitere SubJobs starten (siehe Abbildung 4.4).

Wird der ParentJob abgebrochen, wird die Cancel Message an die SubJobs weiter gegeben. Auf Application Layer ist keine Implementierung notwendig. Terminiert der ParentJob werden etwaige noch laufende SubJobs ebenfalls abgebrochen. Da bekannt ist wieviele SubJobs vorhanden sind, und welche davon bereits terminiert haben, kann der Progress des ParentJobs bestimmt werden. Dieses Konzept sollte auch bei P2P Network Overlays anwendbar sein.

JavaScript Promises unterstützen ein ähnliches Konzept. Promise.race wird für den Fall, dass eine erfolgreiche sub Promise ausreichend ist verwendet und Promise.all für den Fall, dass die erfolgreiche Ausführung aller sub Promises obligat ist.

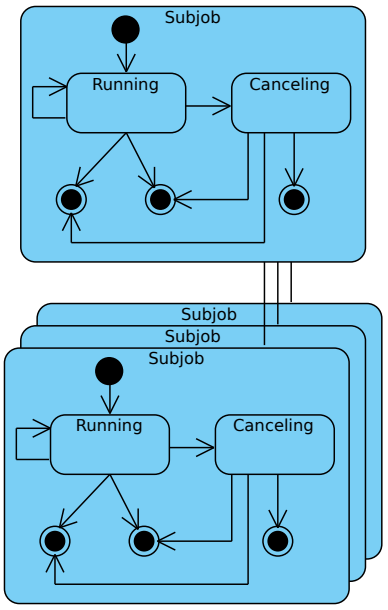

Abbildung 4.3: Immer wenn eine der untergeordneten State Machines terminiert, wird der Workflow Logic Terminate Strategie aufgerufen. Dieser entscheidet ob die Übergeordnete State Machine ihren State ändern muss.

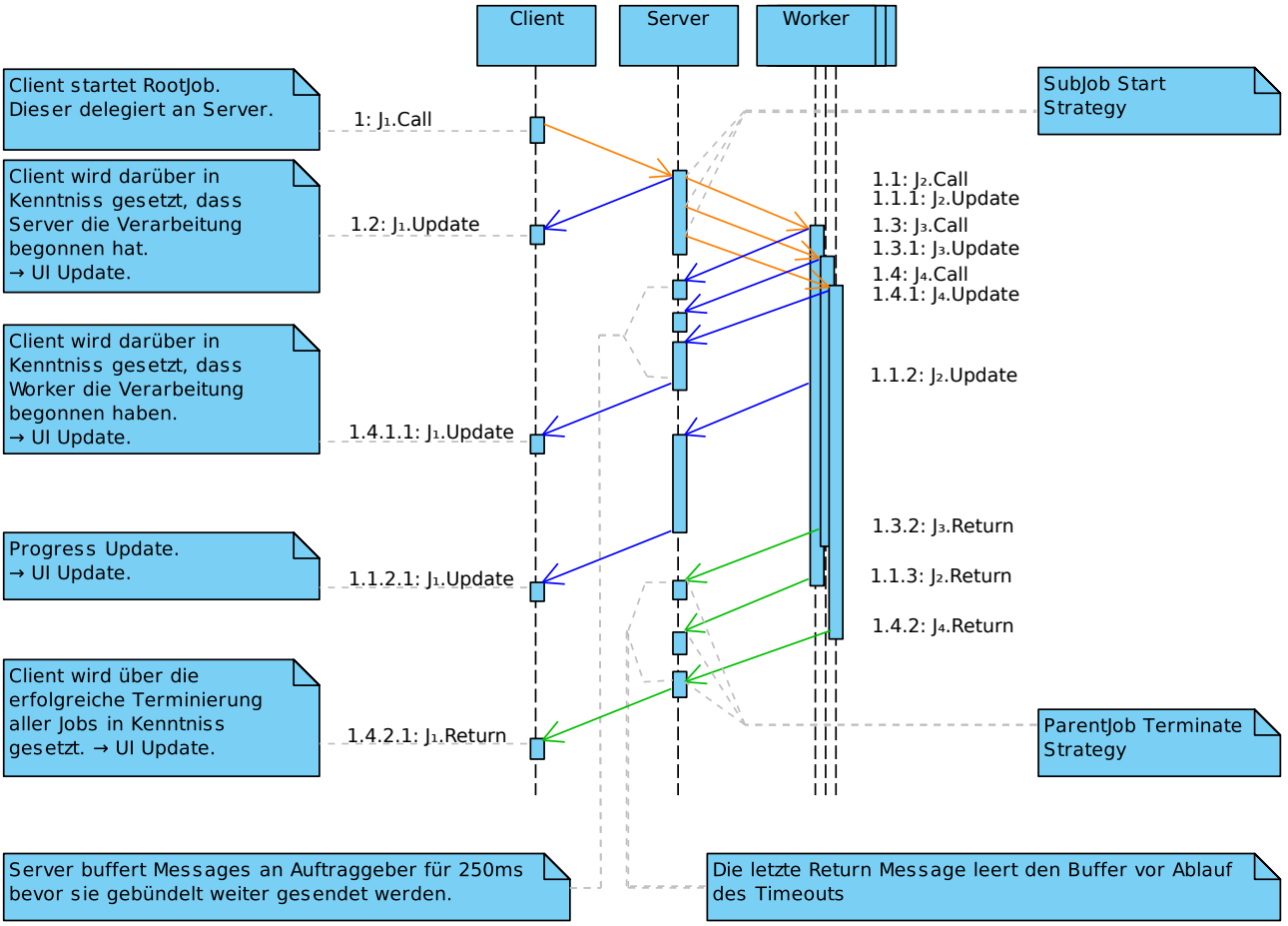

Abbildung 4.4: UML Sequenz Diagramm eines Parallel Workflows. Einfachster Fall ohne Fail oder Cancel Messages. ParentJob terminiert wenn alle SubJobs terminiert haben. J_1 ist RootJob, und zugleich ParentJob von J_2, J_3 und J_4. Orange sind Call Messages, Blau Update Messages und Grün ReturnOk Messages.

Kapitel 5

Implementierung

5.1 Deployment

Um Komponenten ausführen zu können muss Node.js installiert, und *npm install* zum installieren von Third-party Abhängigkeiten verwendet werden. Das Projekt hat folgende Ordnerstruktur:

- data ..... An verteiltes Dateisystem gebunden
- doc
- jobScripts ..... Benutzerdefinierte Scripts
- src
  - app.js ..... Initialisierung und Model
  - config.js ..... Konstanten
  - job ..... Middleware API Library
  - networks
  - views
- server.js ..... *node worker.js* startet einen Server
- worker.js ..... *node worker.js* startet einen Worker
- client.html
- cli.js ..... *node cli.js jobscript.js* führt *jobscript* aus

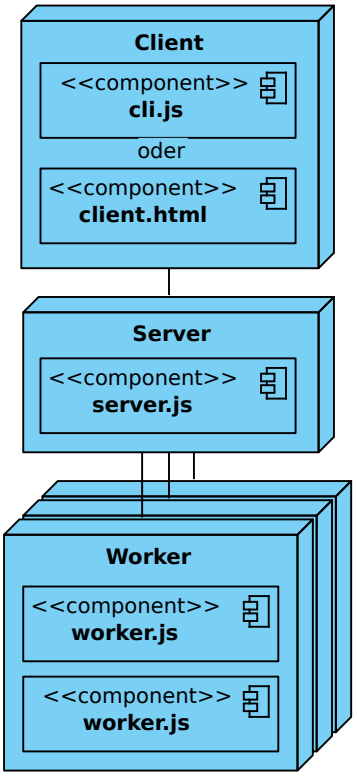

Abbildung 5.1: UML Deployment Struktur mit Netzwerk Verbindungen.

Die System besteht aus mehreren ausführbaren Komponenten: Der Server **server.js**, die Worker **worker.js**, der Webclient **client.html** sofern der Server läuft, und der CLI-Client **cli.js** (siehe Abbildung 5.1). Bis auf den Webclient sind sie alle als Node.js Script startbar. Im Folgenden werden Instanzen dieser als Node bezeichnet. Der CLI-Client erwartet ein auszuführendes JobScript als Argument. Die Gemeinsamkeiten dieser ausführbaren Komponenten werden in Abschnitt 5.2.3 beschrieben.

Die Unterschied dieser Komponenten sind gering: CLI-Client und Webclient unterscheiden sich von den anderen Komponenten nur durch das zusätzlich enthaltene UI. Die Worker unterscheiden sich vom Server nur dadurch, dass sie ClientWebSockets anstatt ServerWebsockts verwenden. Dieses Design wurde gewählt, um zukünfigte arbeiten mit P2P und HCSNO mit mehreren Server und Worker Ebenen zu erleichtern. Der **src/networks/** Ordner könnte mehrere Network Implementierungen enthalten, dezeit ist aber nur eine HCSNO Implementierung mit einer Worker Ebene vorhanden. Das Network Interface wird in 5.2.2 beschieben.

Der **src/job/** Ordner enthält Libary Code. Dieser wird bei der Implementierung von JobScripts benötigt. Enthalten sind die Job Klasse, Workflow Logic Implementierungen, die RemoteJob Klasse und Adapter Jobs [GHJV05] für standard Technologien wie AJAX und das Betriebsystem Prozess API.

5.2 Globale Objekte und deren Initialisierung

Der Browser bzw. Node.js laden zuerst **src/app.js**, eines der Netzwerke aus **src/networks/** und optional ein UI aus **src/views/**. Ist der Ladevorgang abgeschlossen aktiviert die Initialisierungsfunktion *app.init* die geladenen Konponenten in folgenden Schritten:

1. Konstanten aus **src/config.js** werden ins Model übernommen
2. Network Events werden an app gebunden (siehe 5.2.2)
3. Network *connect(ipEndpoint)* wird ausgeführt, die eigene Node Info an den Server gesendet, und der Server antwortet mit den Node Infos aller anderen Nodes. Der Endpoint wird aus app.config genommen.
4. Es werden Timer gestartet um NetInfo aktuell zu halten (siehe 5.2.3.4)
5. Eine Liste der am Server verfügbaren JobScript wird geladen
6. Der CLI-Client führt an dieser Stelle das als Argument gegebene JobScript aus
6. Der Webclient bindet GUI Elemente an das Model

5.2.1 View

Views sind optional. CLI- sowie Webclient UI Elemente implementieren das selbe Interface. Dieses besteht nur aus einer *update(diff)* Funktion. Das Argument *diff* enthält neue, gelöschte und geänderte Teile des Models. Die update Funktionen werden ausschießlich von Model *onChange* Events aufgerufen.

5.2.2 Network

Die geladenen Netzwerke sind zunächst passiv. Sie stellen folgendes Interface zur Verfügung:

- Die *connect(ipEndpoint)* Funktion initialisiert aktiv eine Verbindung
- Das *onConnect(netInfoDiff)* Event wird ausgelöst wenn eine Verbindung aufgebaut wurde
- Das *onDisconnect(netInfoDiff)* Event wird ausgelöst wenn eine Verbindung terminiert
- Das *onMessage(data)* Event wird ausgelöst wenn eine Message empfangen wird

5.2.3 App

Wird eine Message empfangen, wird zunächst der zugehörige Job anhand der JobID und der RunningJobs Lookup Table ermittelt, und danach die durch die Message definierte Transition ausgeführt. Die Job Event Funktionen modifizieren dabei das Model (siehe Tabelle 5.1). App enthält auch das Model welches sich aus *Mergeable* Komponenten zusammensetzt. Diese werden ausschließlich mit ihrer *update(diff)* Funktion modifiziert, und lösen dabei ihr *onChange* Event aus. Damit kann das UI aktuell gehalten werden. Das Model setzt sich aus folgenden *Mergeable* Komponenten zusammen:

5.2.3.1 Config

Enthält Konstanten aus **src/config.js**, erweitert diese mit der Mergable Funktionalität um an das UI gebunden werden zu können.

5.2.3.2 JobScripts

Enthält eine Liste der am Server bekannten JobScripts. Die Script Inhalte werden erst bei Zugriff geladen. Jede UI Action erstellt einen Job basierend auf dem im JobScript definirten Prototypen.

5.2.3.3 RunningJobs Lookup Table

RunningJobs ist eine Liste aller Jobs die auf dem Gerät gerade ausgefügrt werden, und aller noch laufenden RemoteJobs die auf diesem erstellt wurden. Sie ist zu Beginn leer, Call Transitions tragen Jobs ein, Return Transitions entfernen sie.

5.2.3.4 NetInfo

Als member des globalen Models, steht diese Komponente in allen Modulen zur Verfügung, auch in JobScripts. Das NetInfo Objekt wird in JobScripts verwendet um geeignete Worker auszuwählen. Es besteht aus einer Liste der im System vorhandenen Nodes. Jede Node darin bietet folgende Members:

- Funktionen zum Empfangen und Senden
- Momentane CPU- und Speicherauslastung
- Informationen über das Betriebssystem
- Liste der auf der Node installierten Interpreter

NetInfo ist zu Beginn leer, und wird durch Network Events *onConnect* und *onDisconnect* sowie zyklischen Messages aktuell gehalten. Worker senden zyklisch ihre NetInfo Struktur an den Server. Das ist ausreichend um das NetInfo Model am Server aufzubauen. Der Server sendet zyklisch die akkumulierten Änderungen an die Clients.

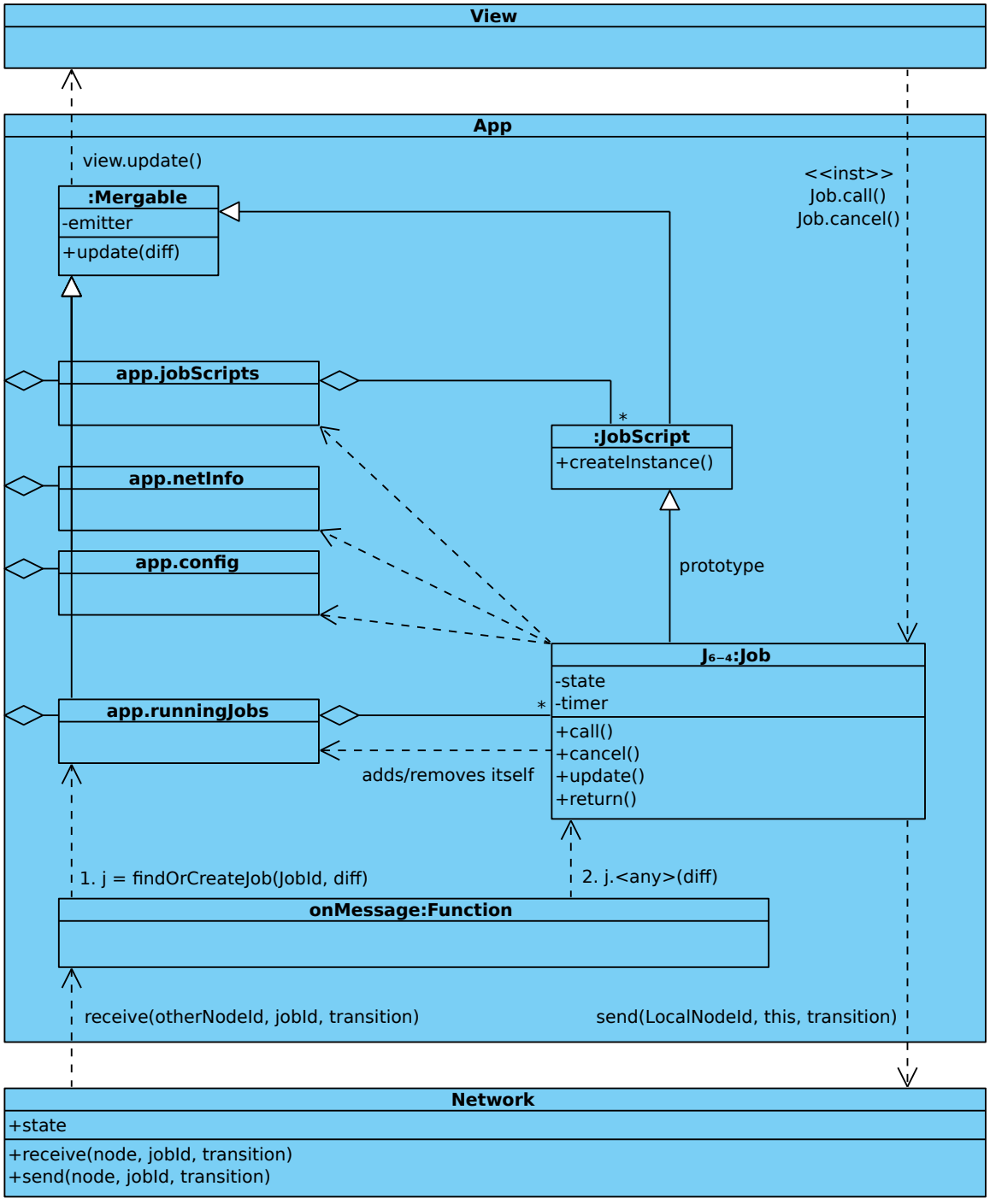

Abbildung 5.2: UML Object Diagramm des Client. Server and Worker enthalten keine View.

5.3 Job API

Die zentrale Klasse des Middleware API ist die *Job* Klasse. Zu finden in **src/job/**. Sie deckt die Funktionalität von JavaScript-Promises ab, und erweitert diese um Abbrechbarkeit und optionale Zwischenergebnisse. Jeder Job kann mit einem Timeout versehen werden, dies erleichtert die Implementierung vom RemoteJobs, und kann auch bei lokalen Jobs hilfreich sein.

Die abstrakte Job Klasse implementiert die in Kapitel 4.2 beschriebene State Machine. Für den Anwender bedeutet dies, dass sie nicht erlaubte Transitions abfängt. State Transitions werden mit den Funktionen call, cancel, update und return ausgelöst. Zu jeder Transition kann der Anwender eine Funktion definieren, die mit der Transition ausgeführt wird. Im Folgenden werden diese Funktionen Job Event genannt. Die Implementierungen dieser Funktionen werden an den Job Konstruktor übergeben. OnCall muss definiert werden, onCancel, onUpdate und onReturn sind optional (siehe Abbildung 5.3).

```
1
2  var j = new Job({
3    params:{},
4    onCall:  j=> {
5      // calculate something
6      j.update(0.5, 'half done')
7      // calculate result
8      j.return('ok', 'done')
9    },
10   onUpdate:j=> print(j.output)
11   onReturn:j=> print(j.output)
12 })
13 j.call()
14
```

Abbildung 5.3: Erstellen und starten eines einfachen lokalen Jobs.

Die bei JavaScript Promises an den Konstruktor übergebene Funktion entspricht dem onCall Event des Jobs. Die Job.return Funktion übernimmt die Aufgaben der reject und resolve Funktionen von Promises. onCancel und onUpdate existieren bei JavaScript Promises nicht, da diese nicht abbrechbar sind und keine Zwischenergebnisse liefern können.

Jeder Job muss terminieren, ähnlich wie Promises mit reject oder resolve es tun. Es gibt vier Arten wie eine onCall Implementierung dies tun kann: Durch einen Aufruf der Job.return Funktion, eine Exception, ein Timeout, oder durch binden des Jobs an SubJobs mit Hilfe der Job.delegate Funktion (siehe 5.5, 4.3). Verwendet man Job.onReturn kann neben dem Ergebnisszustand auch ein Ergebnisswert mitgegeben werden. Für die Übergabe von Zwischenergebnissen und den Progress wird Job.update beliebig oft verwendet. Ist der Job an SubJobs gebunden, wird Job.return von der Workflow Logic aufgerufen.

Event Auslöser 5.3	Message (nur bei RemoteJobs) 4.1	Transition 4.2	Job Event Funktion 5.3
Job.call(args)	Call	call	Job.onCall(param)[1]
Job.cancel()	Cancel	cancel	Job.onCancel()
Job.update(p, d)	Update	update	Job.onUpdate(p, d)
Job.return('ok', rv)	Return	returnOk	Job.onReturn(s, rv)
Job.return('canceled')	Return	returnCanceled	Job.onReturn(s)
Job.return('fail', e) ∨ Exception ∨ Timeout	Return	returnFailed	Job.onReturn(s, e)

Tabelle 5.1: Links: Die vom Anwender aufgerufen Steuerfunktionen, rechts die vom Anwender definierten Event Handler. Nur RemoteJobs verwenden Messages. Transitions lösen Job Events aus. [1] obligates Konstruktor Argument der Job Klasse.

5.4 Vorgefertigte Jobs

Die in 5.3 beschriebene Job Klasse ist abstrakt und dafür vorgesehen vom Anwender spezialisiert zu werden. Dieser Abschnitt beschreibt drei in der Middleware enthaltenen Spezialisierungen.

5.4.1 RemoteJob

RemoteJobs führen die onCall und onCancel Funktion auf einem Remote Gerät aus. Dem Konstruktor wird eine Node aus dem NetInfo Objekt übergeben welche definiert wo sie ausgeführt werden.

5.4.2 AjaxJob

Adaptiert XMLHttpRequest damit dieser in Job Trees eingesetzt werden kann. OnCall löst den Request aus. ReadyState 1 bis 3 löst update Events aus, ReadyState 4 das onReturn Event [W3C14].

5.4.3 OsProcessJob

Adaptiert das Node.js Prozess Interface, damit OS-Prozesse in Job Trees eingesetzt werden können. OnCall startet den Prozess, onCancel sendet ein SIGTERM. Stdout wird auf Job.update umgeleitet.

5.5 Implementierung von JobScripts mit Job Trees

JobScripts werden im **src/jobScripts** Ordner abgelegt. Der RootJob wird immer vom UI erzeugt. JobScripts enthalten nur die onCall Funktion eines Jobs und optional default parameter. Das UI definiert die onUpdate und onReturn Funktionen, um Ergebniszustand und Daten anzuzeigen.

Abbildung 5.4 zeigt ein Script, dass 20 'leere' Jobs auf Worker Nodes ausführt. Die Skript Funktion enthält Code von Client, Server und Worker. Das onCall Event erhält den RootJob *j* als parameter. Er wird nicht direkt abgearbeitet, sondern an einen RemoteJob auf dem Server delegiert. *j.delegate* bindet den Erfolgszustand von *j* an den des RemoteJobs *js* mit Hilfe der toOne Workflow Logik. Anders betrachtet fügt es Knoten in den Job Tree ein. Derzeit unterstützt *Job.delegate* folgende Workflows: toOne, parallel, pool und konsekutiv. Bis auf toOne fügen alle anderen Workflows mehrere Knoten hinzu. Deshalb ist das *job* Argument der delegate Funktion als Factory Method [GHJV05] ausgeführt.

```
1
2  onCall: j=> j.delegate({
3      type: 'toOne',
4      job: ()=> new RemoteJob({
5          desc: 'init workers on server',
6          node: app.netInfo.server,
7          args: j.params,
8          onCall: js=> js.delegate({
9              type: 'pool',
10             workerPool: app.netInfo.filter('POSIX64'),
11             jobCount: 20,
12             job: (idx, poolNode)=> new RemoteJob({
13                 desc: 'empty job on worker',
14                 node: poolNode,
15                 args: {},
16                 onCall: jw=> jw.ret('ok', 'no result')
17             })
18         })
19     })
20 })
21
```

Abbildung 5.4: Ein Pseudo JobScript das 20 leere Jobs auf Workern ausführt. Blau: Client, Grün: Server, Rot: Worker.

Es ist nicht verpflichtend wie in diesem Beispiel durchgängig RemoteJobs zu verwenden. Sie sollten nur Anwendung finden wenn es notwendig ist auf ein anderes Gerät zu wechseln. Das onCall Event von RemoteJobs wird auf der Node ausgefürt die im Konstruktor spezifiziert wurde. RemoteJobs bilden im Code Geräteübergänge. Abbildung 5.4 zeigt Farblich auf welchen Geräten der Code ausgefürt wird, und nicht die Jobgrenzen.

Man beachte, dass *js.delegate* am Server einen Pool Workflow verwendet, also mehrere WorkerJobs (zugleich RemoteJobs in 5.4) ausgeführt werden, die sich nur durch ihre Node und Parameter unterscheiden. 5.4 Zeile 10 zeigt die Auswahl der Worker die den Pool bilden. *netInfo.filter* ist eine Funktion die alle 64bit Posix kompatiblen Nodes aus der NetInfo Datenstruktur liefert. *jw* Terminiert bei diesem einfachen Beispiel unmittelbar nach dem Start. Realistischere Beispiele binden an dieser Stelle einen OsProcessJob ein, oder implementieren einen Nutzlast Algorithmus direskt in JavaScript.

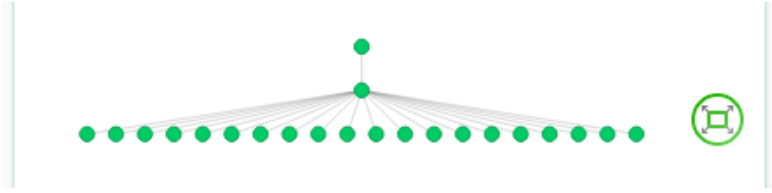

Abbildung 5.5: Screenshot der Job Tree Visualisierung des Webclients. Jeder Knoten ist ein Job. Grün zeigt den Endzustand Ok an. Der gezeigte Job Tree wurde mit dem Script aus Abbildung 5.4 erzeugt. **Oben**: Der am Client vom UI erzeugte RootJob (blau in Abbildung 5.4). **Mitte**: Der am Client erzeugte, und am Server ausgeführte RemoteJob (grün in Abbildung 5.4). **Unten**: Die am Server erzeugten, und auf den Workern ausgeführten RemoteJobs (rot in Abbildung 5.4.)

Verwendet man *Job.delegate*, kann die Middleware Meta Informationen über den Ablauf (Job Trees) sammeln. Die in den folgenden Kapiteln gezeigten Visualisierungen basiren auf Job Trees.

Kapitel 6

Evaluierung

Dieses Kapitel evaluiert den *pool* und *parallel* Workflow mit mehreren JobScripts. Der Webclient wird verwendet um einzelne Test Jobs auszuführen, das CLI um Test Jobs 25 mal auszuführen. Gemessen wurden neben der Gesamtlaufzeit (RootJob Laufzeit) auch die Laufzeiten der WorkerJobs. Joblaufzeiten werden immer auf dem Rechner auf dem der Job erstellt wurden gemessen, für RemoteJobs bedeutet das inklusive des Netzwerk Roundtrips. Das System wird mit beiden Clients einem Stresstest unterzogen, indem WorkerJobs mit minimaler Laufzeit (sie terminieren sobald das onCall Event aufgerufen wird) eingesetzt werden (6.1.3, 6.2.2). Dieser Stresstest wird mit dem Pool Workflow durchgeführt, da er mehr Netzwerk Messages benötigt.

Ziel der CLI Messungen ist es zu zeigen, dass das JobScript skaliert (6.1.1, 6.1.2). Jeder Test führt einen Job auf 1, 2 und 4 Worker Devices aus. Auf jedem Worker Device laufen 4 Worker Nodes. Verteilte Systeme sind nicht zeitlich deterministisch, desshalb werden alle Messungen 25 mal wiederholt. Die Skalierbarkeit wird mit einem Boxplot der Gesamtlaufzeit mit linearen Achsen, und einem mit logaritmischen Achsen veranschaulicht.

Webclient Tests führen ihren Test Job nur ein mal aus. Der Webclient ist in der lage Gantcharts aus Job Trees zu generieren. Diese werden genutzt um die Probleme aufzuzeigen die beim implementieren von JobScripts entstehen können (6.2.1, 6.2.2).

Webclient Tests						
Abschnitt	Wiederh.	Workflow	WorkerJobs	Worker Nodes	Worker Devices	mean(T) [ms]
6.1.1 👍	25	Parallel	4	4	1	7 004
			8	8	2	3 706
			16	16	4	2 342
6.1.2 👍		Pool	65	4	1	672 136
				8	2	332 253
				16	4	154 961
6.1.3		Pool	20	4	1	103
				8	2	120
				16	4	118
CLI Tests						
Abschnitt	Wiederh.	Workflow	WorkerJobs	Worker Nodes	Worker Devices	T [ms]
6.2.1	1	Pool	12	3	1	21 209
6.2.2		Pool	20	3	1	1 064

Tabelle 6.1: Test Übersicht. Grüner Daumen markiert lineare Skalierbarkeit.

Als Hardware standen fünf Rechner mit gleicher Ausstattung, jeweils 4 Cores und 16GB RAM, zur Verfügung. Pro Rechner wurden maximal vier Worker verwendet um Swapping zu vermeiden, denn die JobScripts verwenden nur einen primitiven Scheduler. Als Betriebssystem wurde Debian Linux verwendet. Experiment 6.1.2 startet WorkerJobs die auf ein verteiltes Dateisystem (AFS) zugreifen. AFS cached die Lesevorgänge.

6.1 CLI Tests

6.1.1 Primzahlen Suche - Parallel Workflow

Dieses Experiment wurde ausgewählt, weil es einen sehr einfachen Fall zeigt, der auch gut skaliert. Der Algorithmus duchsucht den Bereich 10^7 bis $2 \cdot 10^7$ nach Primzahlen und übergibt die Anzahl der gefundenen Primzahlen zusammen mit dem Progress in Echtzeit an den Client. Die WorkerJobs sind in C++ implementiert, und benötigen keinen Dateisystem IO. Für Laufzeiten siehe Tabelle 6.1.

Der Serverteil des JobScripts teilt den zu durchsuchenden Bereich proportional auf die zur Verfügung stehenden Worker auf. Somit ist die Anzahl der WorkerJobs immer gleich der Anzahl der Worker. Stehen mehr Worker zur Verfügung, wird der zu durchsuchende Bereich für jeden Worker kleiner. Histogramm 6.1 zeigt, dass mittlere WorkerJob Laufzeit annähernd linear mit der Anzahl der verfügbaren Worker sinkt. Die Gesamtlaufzeit zeigt die erwartete lineare Skalierbarkeit, ersichtlich in Abbildung 6.2 und 6.3.

Das Messaging Verhalten zwischen Server und Worker benötigt nur einen Roundtrip je Worker Node. Zu begin sendet der Server eine Call Message an jede Worker Node, diese senden im 250ms Intervall Update Messages an den Server, und am Ende möglichst gleichzeitig jeweils eine Return Message.

Der parallel Workflow skaliert nur dann linear, wenn alle WorkerJobs gleich große Laufzeiten aufweisen. Bei der Primzahlensuche ist dies im Detail betrachet nicht der Fall, weil großere Zahlen eine größere Laufzeit veruhrsachen und die Zahlen im letzten Block größer sind.

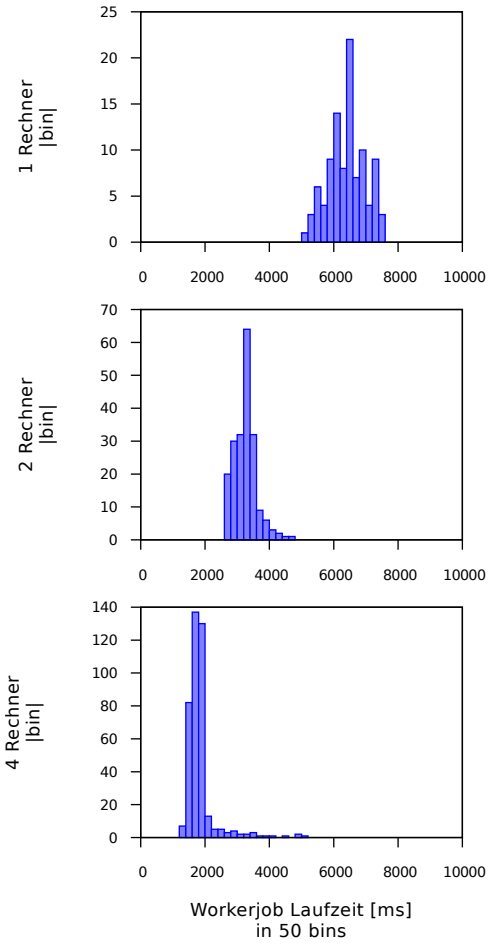

Abbildung 6.1: Experiment 6.1.1 WorkerJob Laufzeiten aller 25 Iterationen in 50bins.

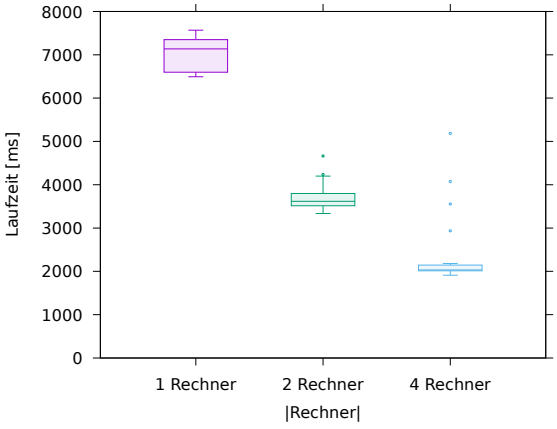

Abbildung 6.2: Experiment 6.1.1. Boxplots für Gesamtlaufzeit mit 1, 2, und 4 Worker, und je 25 Iterationen

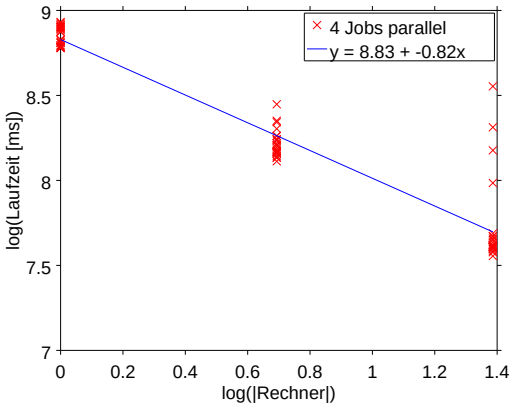

Abbildung 6.3: Experiment 6.1.1 Gesamtlaufzeiten mit 1, 2, und 4 Workern, und je 25 Iterationen. X und Y Achse logaritmiert.

6.1.2 Algorithmus zum Vorverarbeiten von 3D Meshes - Pool Workflow

Experiment 7.2 zeigt die verteilte Anwendung eines Algorithmus zum Vorverarbeiten von 3D Meshes. Jeder WorkerJob verarbeitet eine von 65 Input-Dateien mit Hilfe eines C++ Programms. Input- und Output-Dateien werden auf ein AFS Dateisystem geschrieben. Der Progress wird nur erhöht wenn die Verarbeitung einer Datei abgeschlossen wurde. Dieses Experiment unterscheidet sich von Experiment 6.1.1 vor allem dadurch, dass deutlich mehr WorkerJobs erzeugt werden als Worker zur Verfügung stehen. Für Laufzeiten siehe Tabelle 6.1.

Würde man die Anzahl der parallel laufenden WorkerJobs nicht nach oben begrenzen, würde es zu Swapping kommen und die Gesamtlaufzeit verschlechtert sich. Die Begrenzung der Anzahl an parallel laufenden WorkerJobs verhält sich wie Thread Pooling. Theoretisch sollte dieses Script schlechter skalieren als 6.1.1, da nach Terminierung eines WorkerJobs ein Roundtrip notwendig ist um den nächsten WorkerJob zu starten. Insgesamt sind im Vergleich zu Experiment 6.1.1 $n - w$ zusätzliche Roundtrips notwendig, wobei n die Anzahl der WorkerJobs, und w die Anzahl der Worker ist. Dieser Nachteil macht sich in der Gesamtlaufzeit allerdings kaum bemerkbar, solange ein WorkerJob (wie in diesem Fall) viel länger dauert als ein Roundtrip.

Ein weiterer Grund, warum dieses Experiment schlechter skaliert als Experiment 6.1.1, ist die ungleiche Verteilung der WorkerJob Laufzeiten. Wird der WorkerJob mit der längsten Laufzeit als letzter gestartet, wird dieser noch laufen, wenn alle anderen bereits terminiert haben - oder anders gesagt, es werden nicht alle Ressourcen über die gesamte Laufzeit des Algorithmus genutzt (siehe Abschnitt 6.2.1). Sind die Laufzeiten der WorkerJobs im Voraus bekannt, könnte dieses Problem minimiert werden.

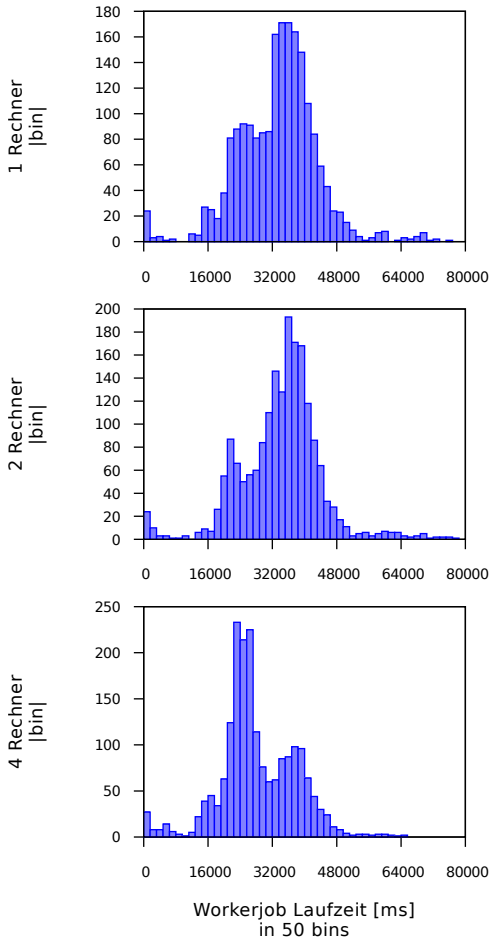

Abbildung 6.4: Experiment 6.1.2 WorkerJob Laufzeiten aller 25 Iterationen in 50bins. Bei Pool Workflow sollte sich das Histogram nicht ändern.

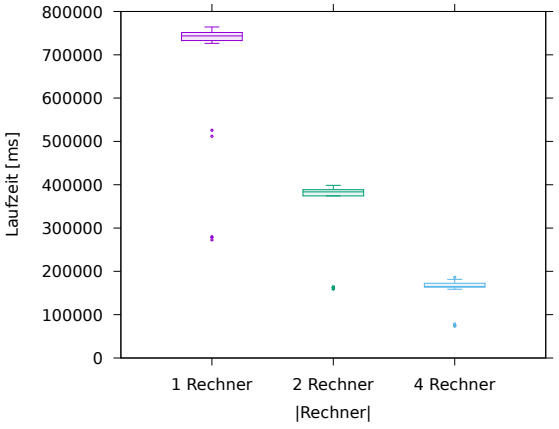

Abbildung 6.5: Experiment 6.1.2. Boxplots für Gesamtlaufzeit mit 1, 2, und 4 Worker, und je 25 Iterationen

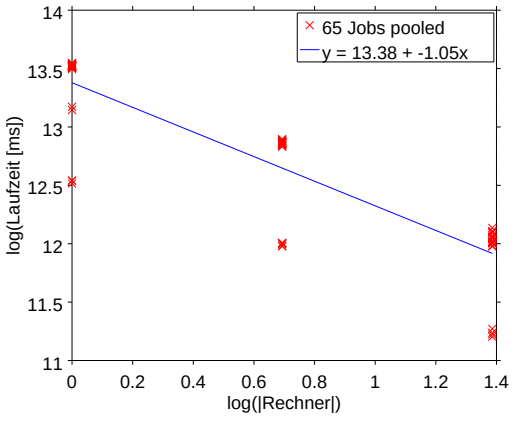

Abbildung 6.6: Experiment 6.1.2 Gesamtlaufzeiten mit 1, 2, und 4 Workern, und je 25 Iterationen. X und Y Achse logaritmiert.

6.1.3 Empty Jobs - Pool Workflow

Dieses Experiment soll den durch die Middleware verursachten Overhead bei Verwendung eines Pool Workflows zeigen. Die im Pool gestarteten WorkerJobs führen keinen Nutz-Algorithmus aus, sie terminieren unmittelbar nach dem Start. Für Laufzeiten siehe Tabelle 6.1. Wie alle anderen Experimente wird auch dieses mit einem, zwei und vier Rechnern mit je vier Workern ausgeführt.

Es zeigt sich, dass der Server bereits mit acht Workern, überlastet ist. Zu erkennen ist dies am Histogramm in Abbildung 6.7. Die Laufzeit von RemoteJobs wird am auftraggebenenden Gerät gemessen, inklusive Roundtrip, Serialisierung, und Workflow Logic. Das Histogramm zeigt schon bei acht, und vor allem bei 16 Workern viel längere Laufzeiten. Da die tatsächliche Laufzeit am Worker nahezu null ist und angenommen werden kann, dass die Roundtrip Zeit sich nicht wesentlich veränert hat, muss die zusätzliche Laufzeit dem Server zugeschrieben werden.

Die Message Sequenz ist die selbe wie bei 6.1.2. Gleicher Workflow bedeutet immer gleiche Message Sequenz mit Ausnahme der Update Messages, die hier vernachlässigt werden können. Die Zeitabstände sind aber geringer, da die WorkerJobs sofort Terminieren. Erhält der Server von zu vielen Workern Return Messages in kurzen Abständen, wächst die Queue am Server. Die Auslastung des Servers ist von der Anzahl der Worker, die mit ihm verbunden sind, linear abhängig. Ein HCSNO mit mehreren Worker Ebenen könnte dieses Problem beheben, da es die Serverauslastung logarithmisch von der Gesamtanzahl der Worker abhängig macht. Ein solches System könnte in einer zukünftgien Arbeit untersucht werden.

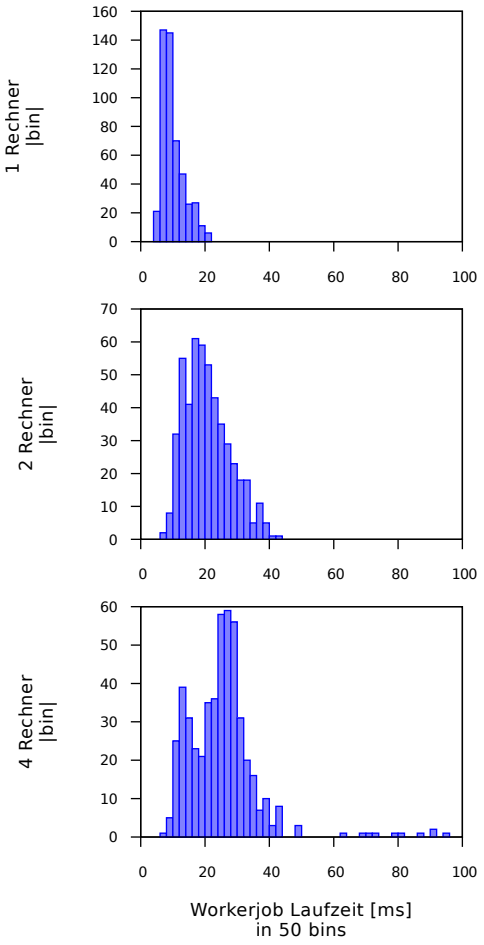

Abbildung 6.7: Experiment 6.1.3 WorkerJob Laufzeiten aller 25 Iterationen in 50bins. Bei Pool Workflow sollte sich das Histogram nicht ändern.

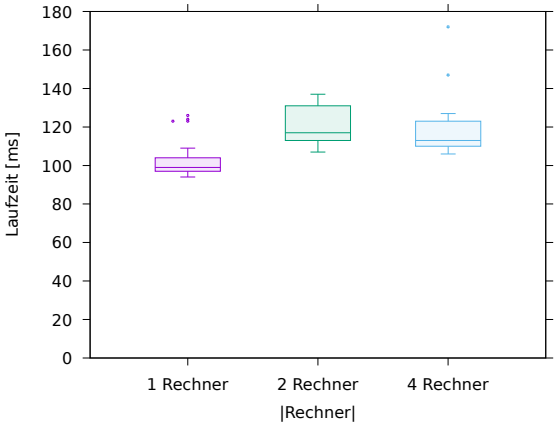

Abbildung 6.8: Experiment 6.1.3. Boxplots für Gesamtlaufzeit mit 1, 2, und 4 Worker, und je 25 Iterationen

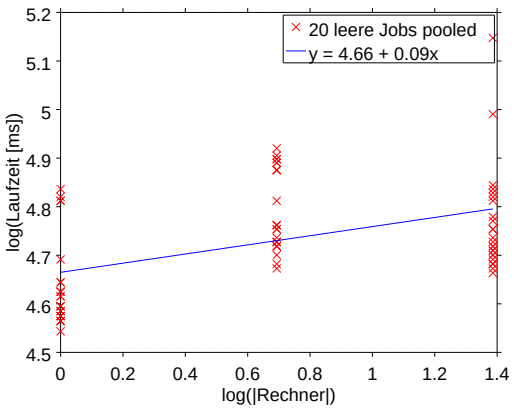

Abbildung 6.9: Experiment 6.1.3 Gesamtlaufzeiten mit 1, 2, und 4 Workern, und je 25 Iterationen. X und Y Achse logaritmiert.

6.2 Webclient Tests

6.2.1 Scheduler Fail

Folgendes Script führt 12 WorkerJobs mit zufälliger Laufzeit zwischen einer und 15 Sekunden in einem Pool von drei Workern aus.

Die Summe der Worker Laufzeiten beträgt 49s, durch die Anzahl der Worker geteilt sollte die Gesamtlaufzeit 17s nicht überschreiten. Die Laufzeit beträgt aber 21s (siehe Tabelle 6.1), das System war also zu weniger als 84% ausgelastet. Abbildung 6.10 zeigt die nicht genutzten Worker Ressourcen im Gant Chart. Ungünsige Konstellationen können eine noch weitaus schlechtere Auslastung zur Folge haben. Bei unregelmäßigen Worker-Job Laufzeiten können andere Scheduler-Implementierungen für eine bessere Auslastung sorgen. Das Scheduler Interface hat derzeit die Einschränkung, dass die Worker Nodes beim Start des Workflows bekannt sein müssen.

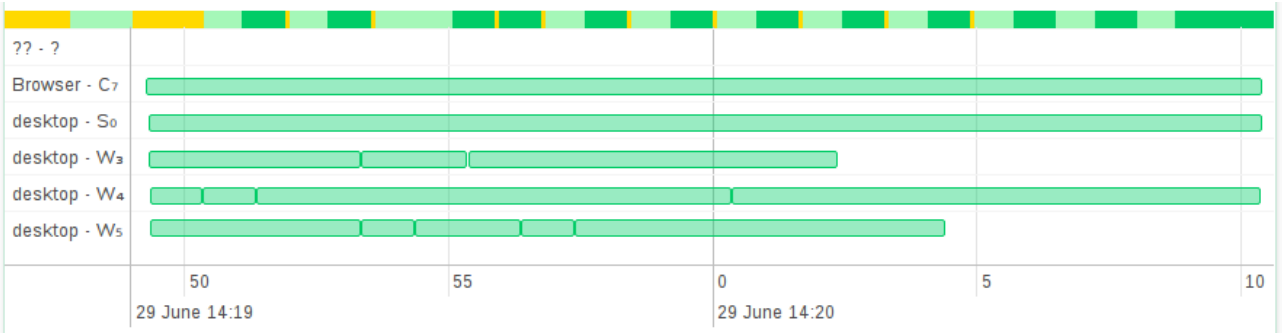

Abbildung 6.10: Gant Chart von 12 Workerjobs mit zufälliger Laufzeit. W_4 verzögert den Abschluss von S_0 und C_7 und verursacht dadurch eine und schlechte Systemauslastung. Diese Abbildung ist ein Screenshot des Webclients.

6.2.2 Client Überlastung

Folgendes Script führt 20 'leere' WorkerJobs in einem Pool von drei Workers aus. Es ist das selbe Script das in Experiment 6.1.3 verwendet wurde. Experiment 6.1.3 hat gezeigt, dass der CLI-Client leere Jobs mit bis zu 4 Worker Nodes verarbeiten kann. Dieses Experiment wird mit 3 Worker Nodes ausgeführt, um eine Überlastung des Servers auszuschließen.

Am Webclient benötigt es 1064ms, der CLI-Client nur 103ms. Der Webclient muss überlastet sein, da der Rest des Systems sich nicht geändert hat. Auf den ersten Blick lässt Abbildung 6.11 vermuten, dass der Server überlastet ist. Berücksichtigt man aber das S_0 am Client erstellt wurde, und Laufzeiten immer auf dem erstellenden Device gemessen werden, kann diese Fehlinterpretation erklärt werden. Der Webclient ist, auch wenn die Anzahl der Worker korrekt dimensioniert wurde, anfällig für Überlastungen da GUI Aktualisierungen rechenintensiv sind.

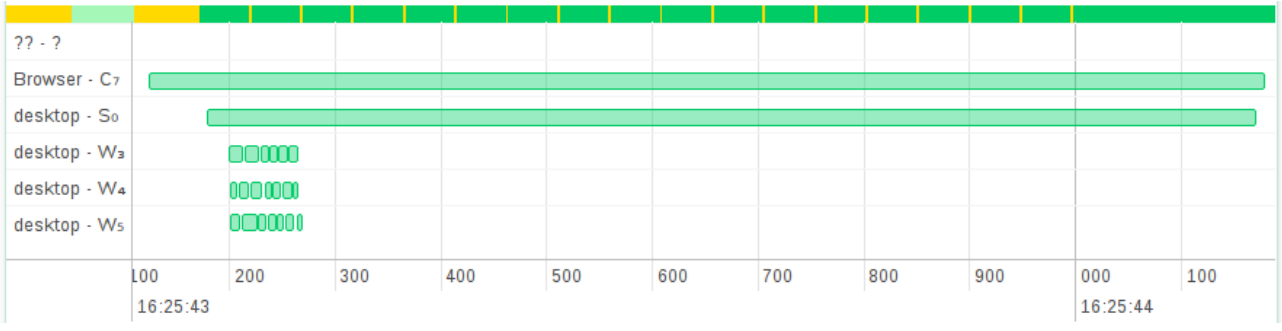

Abbildung 6.11: Gant Chart von 20 Workerjobs mit minimaler Laufzeit. Das Scheduling der WorkerJobs ist akzeptabel. C_7 terminiert spät aufgrund einer Überlastung des Webclients. Diese Abbildung ist ein Screenshot des Webclients.

Kapitel 7

Diskussion

7.1 Fazit

Der folgende Abschnitt diskutiert die Vor- und Nachteile von on demand verteilten JobScripts, der tiefen Position der Middleware im Schichtenmodel, dem asyncronen Job API, dem Job Tree Konzept und der UI Anbindung.

7.1.1 Code on demand

Die Implementierung hat gezeigt, dass die gewählten Technologien gut geeignet sind, um on demand verteilte Scripts von der Middleware ausführen zu lassen. Der on demand verteilte Code eröffnet jedoch Angriffsmöglichkeiten. Ein möglicher Lösungsansatz wären abgesicherte Sandboxes am Server und den Workern, wobei bedacht werden muss, dass das Starten von Prozessen ein gewünschtes Feature ist. Die Verwendung von Secure Websockets, einer Benutzeranmeldung am Client und eine Assoziation der Websocket Verbindung mit einem Benutzer am Server wäre ebenso denkbar. Dabei ist jedoch zu beachten, dass für jeden Benutzer des Webservers auch ein Benutzer im Betriebssystem angelegt werden muss.

7.1.2 Jobs und Promises

Das asynchrone API ist geeignet um Roundtrip arme JobScripts zu schreiben, allerdings auch komplizierter als ein synchrones. Die Ahnlichkeit zu JavaScript Promises könnte zu einer Kompartiblität ausgebaut werden. Jeder Job kann mit einem Timeout versehen werden. Netzwerkprogrammierung erfordert häufig die Berücksichtigung von Timeouts, meist im Zusammenhang mit asynchronen Vorgängen. Die Vereinigung dieser beiden Features in einer Klasse ermöglicht eine kompakte Implementierung der Requirments mit 1316 Zeilen JavaScript. Timeouts finden auch bei lokalen asynchronen Operationen Anwendung. Bei Promises kann man zwar Hardware Fehler vernachlässigen, aber Implementierungsfehler, wie zum Beispiel das fehlende Schließen einer Promise wird nicht erkannt. Eine solche fehlerhafte Implementierung wird bei Jobs ein Timeout auslösen, was die Fehlersuche erleichtert.

7.1.3 Low-level

Das Middleware API ist auf einer tiefen Ebene im Schichtenmodell angesiedelt. Daraus resultiert, dass Scheduler Algorithmen (6.2.1), Vermeidung von Überlastungen (6.2.2) und Workflow Logiken im Application Layer realisiert werden müssen. Um diesen Umstand zu verbessern, könnten weitere Layer darüber geschaffen werden, zum Beispiel nach dem Vorbild von MapReduce oder Apache Spark.

7.1.4 Job Trees

Mit der Hilfe der Job.delegate Funktion können Abhängigkeiten zwischen den Jobs deklariert werden. Die Middleware kann so zur Laufzeit einen Job Tree aufbauen, der zu Debugging und Usability Zwecken verwendet werden kann. Zum Beispiel ist im Fehlerfall daraus ersichtlich, auf welchem Gerät der Fehler stattgefunden hat. Auch Ablaufvisualisierungen und Performance-Analysen sind damit möglich.

7.1.5 UI Anbindung

Die onUpdate und onReturn Events der RootJobs können für die Anbindung des UI verwendet werden. Die Anbindung ist sehr einfach, da die beiden Events alle durch das Netzwerk veruhrsachten UI Updates abdecken (siehe Abbildung 4.4). Das gilt für das Anzeigen von Ergebnissen genauso wie für die Anzeige des Progress, Enablen/Disable Events und Fehlermeldungen - egal ob lokal oder remote. Cancel Implementierungen sind vereinfacht auf das triviale Auslösen eines Abbruchs mit einem Funktionsaufruf. Fehlermeldungen können ebenfalls im UI dargestellt werden, indem im onReturn Event auf die Zustände der SubJobs geachtet wird.

7.2 Zukünftige Arbeiten

Es wurden JobScripts für zwei Network Overlays implementiert, Client Server und HCSNO mit einer Worker Ebene. Für diese Overlays wurden jeweils mehrere Test Scripts mit geringem Zeitaufwand implementiert. Das Konzept könnte Anwendung bei der Implementierung von Prototypen finden. Die Network Overlay Struktur ist vorerst aber noch statisch zur Runtime und eine Änderung würde die Anpassung von Kernkomponenten erfodern. Eine Erweiterung um ein P2P Network Overlay würde einen interessanten Raum für Experimente schaffen.

Kapitel 6 zeigt die Skalierbarkeit verschiedener Algorithmen verteilt auf ein HCSNO mit einer Worker Ebene. Es ist ein grundlegendes Problem, dass ein Server nur eine begrenzte Anzahl von Workern bedienen kann. Die JavaScript Implementierung kann im worst Case Vier Worker pro Server bedienen (siehe 6.1.3). Eine sehr große Anzahl von Workern ist nur möglich, wenn nicht alle Worker mit demselben Server verbunden sind. Ein hierarchisches Netzwerk mit mehreren Ebenen könnte für das Starten und Stoppen von Prozessen auf Worker eine Laufzeit $T(n) = \mathcal{O}(log_w(n))$ erreichen, wobei n die Gesamtanzahl der Worker und Server ist, und w die Anzahl der Worker pro Server. Auch die Progress-Anzeige konnte dabei erhalten bleiben.

Derzeit enthält der Webclient mehrere Visualisierungen: den Workflow Graph zu jedem Run eines Scripts, einen Gant Chart, der die Laufzeiten aller Jobs zeigt, sowie eine vereinfachte Darstellung des Netzwerks. Sie alle visualisieren dasselbe Model, den Job Tree. Für dieses Model könnten weitere Visualisierungen geschaffen werden, die bei Analysen helfen würden. Der Job Tree könnte auch verwendet werden um mit Hilfe von Klassifikationsverfahren Fehler und Abweichungen zu erkennen. Abbildung 6.4 zeigt, dass einige WorkerJobs praktisch sofort termininierten, was auf ein Fehlverhalten hinweist.

Das Job API kann noch weiter vereinfacht werden. Eine Kompatibilität zu JavaScript Promises würde bedeuten, dass Anwender die mit diesem schon vertraut sind, das Job API schnell erlernen würden. Eine Trennung des Jobs in zwei Komponenten, Future und Promise nach [BJH77], scheint aber auch sinnvoll.

Literaturverzeichnis

[1&16] 1&1: Qooxdoo Libary. *http://qooxdoo.org* (2016). 3

[Asy16] Async Project: Async javascript libary. *http://caolan.github.io/async* (2016). 9

[BJH77] Baker Jr H. C., Hewitt C.: The incremental garbage collection of processes. In *ACM Sigplan Notices* (1977), vol. 12, ACM, pp. 55–59. 3, 24

[BK03] Bishop T. A., Karne R. K.: A survey of middleware. In *Computers and Their Applications* (2003), pp. 254–258. 3

[FCO99] Falkner K. K., Coddington P. D., Oudshoorn M. J.: Implementing asynchronous remote method invocation in java. In *Proc. the Parallel and Real-Time Systems Conference* (1999). 3

[FK03] Foster I., Kesselman C.: *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003. 4

[Fos06] Foster I.: What is the grid? a three point checklist, july 2002. *ThreePoint-Check. pdf* (2006). 4

[GHJV05] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2005. 9, 11, 15

[Nod16] Node.js Foundation: Nodejs about. *https://nodejs.org/en/about/* (2016). 6

[SK11] Sadashiv N., Kumar S. D.: Cluster, grid and cloud computing: A detailed comparison. In *Computer Science & Education (ICCSE), 2011 6th International Conference on* (2011), IEEE, pp. 477–482. 4

[SSRB13] Schmidt D. C., Stal M., Rohnert H., Buschmann F.: *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, vol. 2. John Wiley & Sons, 2013. 4

[Tan] Tanenbaum S.: Computernetzwerke. *Pearson Verlag GmbH, München*. 8

[W3C14] W3C: Xmlhttprequest level 1. *https://www.w3.org/TR/XMLHttpRequest/* (2014). 14

[YB05] Yu J., Buyya R.: A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing 3*, 3-4 (2005), 171–200. 9

[ZCD*12] Zaharia M., Chowdhury M., Das T., Dave A., Ma J., McCauley M., Franklin M. J., Shenker S., Stoica I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 2–2. 4

Tabellenverzeichnis

Abbildungsverzeichnis

Index