Diplomarbeit

Entwicklung einer Middleware für verteilte Anwendungen in P2P-Netzen

vorgelegt von

Stephan Fudeus

14. Februar 2005

Technische Universität Kaiserslautern
Fachbereich Informatik
AG Verteilte Algorithmen

Betreuer:	Juniorprofessor Dr. Peter Merz
Zweitgutachter:	Dipl.-Math. Katja Gorunova

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Alle wörtlich oder sinngemäß übernommenen Zitate sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Kaiserslautern, den 14. Februar 2005

Stephan Fudeus

meinen Eltern, die mir ein Studium ermöglicht haben

meinen Freunden, die mir ständig Rückhalt geben

Peter, Katja, Thomas, Mathias, für Fachgespräche, Tipps und Ermöglichung dieser Arbeit

Hannah, für alles

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Der Begriff Peer-to-Peer (P2P) ist heutzutage jedem, der die einschlägigen Nachrichten verfolgt oder sich öfter im Internet aufhält, ein Begriff. Viele verstehen unter P2P dabei jedoch nur ein Synonym für das Tauschen von Musik- und Videoformaten (meist unter Verletzung eines Copyrights).

Der eigentliche Begriff P2P kommt jedoch allein aus der Tatsache, dass Rechner nicht über einen zentralen Dienst kommunizieren, sondern jeder Knoten im Netz gleichzeitig Nutzer und Anbieter ist und alle Rechner potentiell miteinander kommunizieren können. Im Rahmen des sog. Filesharing kann jeder Dienstnutzer bei jedem anderen dessen der Welt zur Verfügung gestellte Dateien lesen.

P2P-Netze bieten allerdings viel mehr als das. Statt dem reinen Austausch von Dateien ist es denkbar, nicht nur den Festplattenplatz miteinander zu teilen, sondern auch die häufig brach liegende Prozessorleistung der am Internet angeschlossen Rechner. Bereits seit dem Jahr 1999 ermöglicht es das Projekt SETI@home [ACK$^+$02], dass ein am Internet angeschlossener PC seine freie Rechenleistung dem Projekt zur Verfügung stellt.

Bei SETI@home erfolgt die Berechnung allerdings wieder nicht im Rahmen eines P2P-Netzes. Rechner, die ihre Ressourcen zur Verfügung stellen wollen, laden Aufgaben-Pakete von einem zentralen Server herunter und senden nach Beendigung der Berechnung ihre Ergebnisse zurück an den Server bzw. die Server-Farm. Die Teilnehmer in diesem Netz sind also nicht gleichberechtigt. Im Rahmen des Projektes BOINC [And04] existiert dafür mittlerweile auch eine Middleware, auf der Projekte entwickelt werden können, die auf zentralen Servern Arbeitspakete zur Verfügung stellen und Ergebnisse sammeln.

Bei dieser Art der Aufgabenteilung ist man darauf angewiesen, dass die Gesamtaufgabe in viele einzelne, leicht zu berechnende Teilaufgaben zerlegt werden kann. Kooperierende Lösungen, d.h. Lösungen, die eine Kommunikation der Teilnehmer der Berechnung untereinander erfordern, sind mit diesem Ansatz nicht möglich.

1.1. Ziel der Arbeit

Das Ziel dieser Arbeit ist es, ein Framework und einen funktionsfähigen Prototypen für eine Middleware zu entwickeln, die es ermöglicht, Berechnungen verteilt in einem Peer-to-Peer-Netz auszuführen.

Dabei sollen verschiedene Arten der Berechnung unterstützt werden. Eine Art der Berechnung sind Aufträge, die massiv parallel berechnet werden können, d.h. solche, die in unabhängige Teilaufträge zerlegt werden können und ohne gegenseitige Kommunikation auskommen. Diese Aufträge können entweder so parametrisiert sein, dass jeder Teilauftrag einen anderen Satz Parameter erhält (z.B. Fraktalberechnung), oder zufallsbasiert, so dass jeder Teilauftrag mit den gleichen Ausgangswerten berechnet wird und unterschiedliche Ergebnisse resultieren. Eine andere Art der Berechnung setzt voraus, dass Knoten, die am gleichen Auftrag arbeiten, auch miteinander kooperieren und durch Nachrichten beispielsweise Teillösungen austauschen (z.B. verteilte evolutionäre Algorithmen).

Alle Knoten in diesem Netz sollen gleichberechtigt sein, d.h. jedem Knoten soll es möglich sein, eigene Anwendungen und Aufgaben im Netz zu verteilen. Den verteilten Anwendungen soll zudem die Möglichkeit gegeben werden, mit anderen Knoten, die an der gleichen Aufgabe arbeiten, kommunizieren zu können.

Alle benötigten Informationen zur Berechnung eines Auftrags (d.h. Parameter, ausführbarer Code der Anwendung, Anzahl der zu berechnenden Fälle) sollen den rechnenden Knoten durch Middlewarefunktionen zur Verfügung gestellt werden, ebenso soll der Rücktransport der Ergebnisse zum Initiator durch die Middleware erfolgen. Der Nutzer soll also an seinem Arbeitsplatz seine verteilte Anwendung starten können und die Ergebnisse schließlich wieder auf seinem Rechner vorfinden.

Die Implementierung soll möglichst modular angelegt sein und nicht auf ein konkretes P2P-Protokoll beschränkt sein, sondern einen leichten Austausch bzw. eine nutzergesteuerte Auswahl von verschiedenen Protokollen ermöglichen. Im Rahmen dieses Prototypen soll ein epidemisches P2P-Protokoll realisiert werden.

Als weitere Funktionalität ist erwünscht, dass auch Rechner, die sich hinter Firewalls befinden, am P2P-Netz teilnehmen können. Dazu können frei zugängliche Knoten (solche ohne Firewall) als Proxy-Server dienen, so dass der komplette Netzwerkverkehr anderer Knoten über diese geleitet wird. Auch diese Proxy-Funktionalität soll transparent durch die Middleware gewährleistet werden.

Um einen Einsatz im Internet zu ermöglichen, soll als Basisprotokoll das InternetProtocol (IP) [Pos81e] verwendet werden. Prinzipiell sollen sowohl UDP [Pos80c] als auch TCP [Pos81f] unterstützt werden, den Anwendungen soll auf jeden Fall die Möglichkeit einer verlustfreien Übertragung gegeben werden.

Um einen hohen Grad an Plattformunabhängigkeit zu gewährleisten und möglichst heterogene Umgebungen unterstützen zu können, soll die Implementierung in der Sprache Java [SUNb] von Sun Microsystems erfolgen.

Fehlermodell

Die Middleware soll darauf ausgelegt sein, in einer realen Umgebung abzulaufen. Dies bedeutet, dass jederzeit mit dem Ausfall von Knoten (*Crash-Fehler*) und mit dem Verlust von Nachrichten (*Omission-Fehler*) gerechnet werden muss. Weiterhin können Knoten aus der Sicht eines anderen Knoten jederzeit aus dem P2P-Netz herausfallen (durch bewusstes Beenden der Middleware oder auch durch Separierung des Netzwerks) und auch wieder hinzukommen (Wiederherstellen der Netzwerkverbindung oder Neustarten der Middleware).

In der aktuellen Stufe nicht betrachtet werden soll die Möglichkeit der absichtlichen oder böswilligen Beeinflussung oder sogar Störung des Ablaufs der Middleware durch teilnehmende Knoten. In einer späteren Ausbaustufe kann auf Sicherheitsaspekte mehr Wert gelegt werden. Diese Problemstellung wird im Rahmen dieser Arbeit allerdings nicht weiter berücksichtigt.

1.2. Gliederung der Arbeit

Das folgende Kapitel wird gezielt benötigte Grundlagen im Bereich der P2P-Netze und epidemischen Protokollen vermitteln. Hierbei werden die Eigenschaften und Vorteile von P2P-Netzen gegenüber dem klassischen Client-Server-Paradigma beleuchtet. Daneben werden verschiedenen Ansätze und bisherige Arbeiten zum verteilten Berechnen vorgestellt.

Im dritten Kapitel wird dann zunächst recht allgemein ein System beschrieben, mit dem die gestellte Aufgabe zu lösen ist. Hierbei wird analysiert, welche Komponenten nötig sind und wie diese miteinander interagieren müssen, um ihrer Anforderung gerecht zu werden. Zu diesen Komponenten wird ein konzeptioneller Ablauf vorgestellt, wie die Kommunikation zwischen verschieden Knoten in einem Netz abläuft und wie Aufträge in diesem Netz verteilt werden können und die Ergebnisse den Initiator erreichen.

Das vierte Kapitel widmet sich dann einer konkreten Implementierung. Hier wird ein System entworfen, dass den Anforderungen des Systementwurfs genügt und konkret die Implementierung der einzelnen Komponenten beschrieben. Dort wird dann auch auf notwendige Details wie Threadsynchronisation und Nachrichtenversand eingegangen.

Im letzten Kapitel wird ein Überblick über das Erreichte gegeben. Es wird dargestellt, was im Rahmen dieser Arbeit entworfen, spezifiziert und entwickelt wurde und welche Eigenschaften die entstandene Middleware hat. Anschließend wird mit anderen Arbeiten auf diesem Gebiet verglichen. Hierbei wird nicht Wert auf die Details der Implementierungen gelegt, sondern versucht, die geplanten Ziele zu gewichten und zu vergleichen, wie diese Ziele umgesetzt wurden.

Im Weiteren wird gezeigt, was mit implementierten Anwendungen im Rahmen von Tests erreicht wurde. Es werden Beispiele gegeben, wie ein dynamisch aufgebautes Netz zu einem willkürlich herausgegriffen Zeitpunkt aussah und welche konkreten Anwendungen im Netz verteilt und berechnet wurden.

Abschließend wird sich diese Arbeit mit möglichen Erweiterungs- und Verbesserungsmöglichkeiten beschäftigen. Diese Arbeit soll nur ein Framework und einen Prototypen definieren. Hierbei werden bewusst einige Abstriche hingenommen, um den begrenzten zeitlichen Rahmen einhalten zu können.

Im Anhang wird auf die Konfigurationsmöglichkeiten der Middleware eingegangen und anhand eines Fallbeispiels eine konkrete Anwendung vorgestellt, welche die Schnittstellen und Möglichkeiten der Middleware nutzt, um verteilt berechnet zu werden.

2. Grundlagen

Im diesem Kapitel sollen einige Grundlagen zum späteren Verständnis der verwendeten Paradigmen, Algorithmen und deren Implementierung gelegt werden.

Im weiteren Verlauf sollen der Einfachheit halber die Ausdrücke Rechner, Knoten und Peer sinngleich verstanden werden.

2.1. Middleware

Middleware bezeichnet anwendungsunabhängige Technologien, die Dienstleistungen zur Vermittlung zwischen Anwendungen anbieten, so dass die Komplexität der zugrunde liegenden Dienste und der Infrastruktur verborgen wird (nach [RMB01]).

In unserem Fall kann Middleware als eine Softwareschicht betrachtet werden, welche die Anwendung vom Betriebssystem entkoppelt und ihr dafür abstrakte, vom konkreten zugrunde liegenden Betriebssystem unabhängige Funktionen zur Verfügung stellt. Eine der ersten Arbeiten auf dem Gebiet ist diejenige von P. Bernstein [Ber96] aus dem Jahr 1996.

Middleware soll üblicherweise für Transparenz sorgen, dazu gehört im Bezug auf verteilte Systeme insbesondere die Verteilungs- bzw. Ortstransparenz. Bei der Verteilungstransparenz soll dafür gesorgt werden, dass Ressourcen unabhängig von dem Ort, an dem sie vorhanden sind, genutzt werden können. Zu diesen Ressourcen gehören primär Speicherkapazität und Rechenleistung. Bei einer herkömmlichen Socket-basierten Entwicklung muss der Nutzer selber einen Zielrechner ausfindig machen, seine Daten dorthin transferieren und dann dort die Ressource nutzen.

Eine große Rolle spielt jedoch auch die Homogenisierung von Programmierschnittstellen. Übergreifend über verschiedene heterogene Systemplattformen, Kommunikationsdienste und auch Hardware entsteht durch die Verwendung einer Middleware ein einheitliches Interface.

Aufgabe von Middleware ist es, die Entwicklung von Anwendungen einfacher zu machen, indem allgemeine Programmierschnittstellen zur Verfügung gestellt werden, die die Verteilung und Heterogenität der zugrunde liegenden Hardware und des Betriebssystems maskieren und Details der tieferen Programmierebene verstecken.

Tanenbaum stellt in [TvS03] verschiedene, mit der Zeit gewachsene Middleware-Modelle vor, um Verteilung und Kommunikation zu beschreiben. Als einfaches frühes Modell führt er „Plan 9“ an, wo alle Ressourcen inklusive I/O-Geräte als Dateien behandelt wurden, welche zum Lesen und Schreiben geöffnet und von mehreren Prozessen gleichzeitig genutzt werden konnten. Ob die Datei entfernt oder lokal vorlag, spielte dabei keine Rolle.

Bei verteilten Dateisystemen herrscht ein ähnlicher aber nicht ganz so radikaler Ansatz vor. Hier werden nur Dateien, die auch tatsächlich Daten enthalten, nicht aber I/O-Geräte durch das verteilte Dateisystem transparent behandelt.

Tanenbaum spricht als nächstes die Einführung von RPC (Remote Procedure Call) an. Hier liegt die Betonung darauf, dass die Netzwerkkommunikation beim Aufruf von Prozeduren verborgen bleibt, deren Implementierung eigentlich auf einem anderen Rechner vorliegt. Die Aufrufparameter werden dabei transparent an den anderen Rechner übertragen, dann die Prozedur ausgeführt und das Ergebnis zurückgeschickt. Dem aufrufenden Anwenderprozess scheint es, als ob die Prozedur lokal ausgeführt wird.

Mit der Einführung der Objektorientierung lag der Wunsch nahe, dass Objekte transparent behandelt werden sollen, unabhängig von dem Ort, an dem sich das Objekt gerade befindet. In der Sprache Java ist dies als RMI (Remote Method Invocation) realisiert und basiert darauf, dass sich jedes Objekt nur auf genau einer Maschine befindet und auf den anderen Maschinen nur Schnittstellen bzw. Stellvertreterobjekte vorliegen. Dieser Stellvertreter (oder auch Stub) transferiert wie auch beim RPC die Aufrufparameter (dies können hier auch entfernte Objektreferenzen sein) zum Zielobjekt auf einem anderen Rechner, ruft die entsprechende Methode auf dem Zielobjekt auf und überträgt das Ergebnis zurück zum Aufrufer.

Allgemein bieten Middlewareplattformen verschiedene High-Level-Dienste an, von transparenten Kommunikationsdiensten und Diensten zur Namensauflösung über Persistenzdienste zu einer verteilten Transaktionsbearbeitung, um mehrere Schritte in einer atomaren Transaktion mit mehreren beteiligten Dienstleistern zu koordinieren. Zahlreiche Formen der Transparenz werden in [Int95] definiert, für eine Schaffung von Transparenz auf diesen Gebieten ist immer eine Middleware verantwortlich.

Beispiele für existierende Middlewareplattformen auf verschiedenen Abstraktionsebenen sind u.a. DCE, DCOM, .NET, RPC, RMI, CORBA, JXTA.

2.2. Verteilte Berechnung und ihre Realisierung

Wikipedia [wik] schreibt dazu:

> „Verteiltes Rechnen (engl. Distributed Computing) ist eine Technik der Anwendungsprogrammierung, bei der eine Applikation nicht nur auf verschiedene Prozesse aufgeteilt wird, sondern auf verschiedene Rechner.“

Ziel ist es dabei meist, die Rechenleistung mehrerer kleiner Systeme zur einem großen System zu bündeln. Zum Einen können somit sonst brach liegende Ressourcen genutzt werden, zum Anderen sind viele kleine Systeme u. U. in der Anschaffung und im Betrieb günstiger als ein Hochleistungssystem.

Existierende Arbeiten

Zur Realisierung von verteilten Berechnungen wurden schon einige Arbeiten durchgeführt, welche in teilweise ganz unterschiedliche Richtungen gehen. Einige „Klassiker" und auch Ansätze, die ähnliche Ziele wie die vorliegende Arbeit haben, sollen hier vorgestellt werden.

Klassisches Parallelrechnen

MPI Softwareunterstützung für verteilte Berechnungen gibt es in zahlreichen Ausprägungen. Für die Ausführung in Multicomputern bzw. Cluster-Systemen wurde das Message-Passing-Interface (MPI) entwickelt. Das MPI ist eine Abstraktionsschnittstelle für die Kommunikation über Hochgeschwindigkeitsnetze in parallelen Systemen. Diese wurde nötig, als die Hersteller der Hochgeschwindigkeitsnetze proprietäre Kommunikationsbibliotheken mit ihren Netzen auslieferten um eine effiziente und parallele Kommunikation zu ermöglichen. Die Einführung von MPI hat diese Schnittstelle wieder zusammengeführt.

PVM Parallel Virtual Machine (PVM) ist ein Softwarepaket, dass die Ausführung von verteilten Anwendungen auf Standard-PCs erlaubt. Mittels PVM können die angeschlossenen Rechner wie ein großer Parallelrechner verwendet werden. Der Nutzer ist allerdings nicht davon befreit, genau zu wissen auf welchen Rechnern er Ressourcen nutzen will. Die zu verwendenden Rechner müssen vor Beginn der Berechnung konfiguriert werden, können jedoch auch noch während der Berechnung verändert werden. Die Parallelisierung beruht darauf, dass die zu verteilende Anwendung in kleinere Tasks aufgespalten werden kann und diese Tasks dann einzeln auf unterschiedliche Rechner geschickt werden.

Ausnutzung von ungenutzten Internet-Kapazitäten

SETI@home Im Gegensatz zu diesen Bibliotheken für hochparallele Systeme gibt es auch andere Ansätze für verteilte Berechnungen. So versucht das Projekt SETI@home [ACK+02] seit Jahren erfolgreich, die ungenutzten CPU-Ressourcen der zahllosen sporadisch an das Internet angeschlossenen privaten Rechner dafür zu nutzen, aufwändige

Berechnung durchzuführen. Im Falle des SETI-Projektes handelt es sich bei diesen Berechnungen um eine Suche nach Kommunikationsmustern in aus dem Weltall empfangenen Radiowellen (daher auch der Name SETI@home: The Search for Extraterrestrial Intelligence). Hierbei müssen diejenigen, die ihre CPU-Zeit zur Verfügung stellen wollen, auf ihrem Rechner einen Client installieren, der bei bestehender Netzwerkverbindung Kontakt zum zentralen SETI-Server aufnimmt und sich ein neues Arbeitspaket abholt. Die Berechnung erfolgt, ohne dass eine Netzwerkverbindung bestehen muss. Neben dem Projekt SETI@home gibt es ähnliche Projekte, die dabei das gleiche Paradigma nutzen, darunter z.B. Folding@home [LSSP02] und distributed.net [dis].

Wie bereits eingangs erwähnt gibt es für Projekte, die dem Verfahren bei SETI@home ähneln mittlerweile auch eine Middleware (BOINC [And04]), die sich um das Anbieten und Verteilen der Softwarepakete kümmert. Das Verfahren bleibt aber das gleiche: Unabhängig agierende Clients bauen regelmäßig am Ende einer Berechnung eine Verbindung zu einem zentralen Server oder einer zentralen Server-Farm auf und laden ihre Ergebnisse hoch und neue Aufgabenpakete herunter. Während der Berechnungszeit ist keine Netzwerkverbindung erforderlich, da zwischen den Clients keinerlei Kommunikation vorgesehen ist und sich die Clients untereinander auch nicht kennen.

Ein großer Unterschied besteht wie man sieht darin, dass bei Parallelisierungsbibliotheken wie MPI und PVM immer auch die Kommunikation zwischen den beteiligten Knoten im Vordergrund steht, man spricht hier von enger Kopplung der beteiligten Prozesse. Bei Projekten wie SETI@home kooperieren die rechnenden Knoten nicht, man spricht von loser Kopplung.

DREAM Die *Distributed Resource Evolutionary Algorithm Machine (DREAM)* entstand im Rahmen eines EU-Projektes zur Schaffung einer offenen, skalierbaren und universellen Infrastruktur und wurde erstmals im Jahr 2000 vorgestellt. Bestehende Hardware soll durch die Nutzung sonst ungenutzter CPU-Zyklen besser ausgenutzt werden. Sie soll den Nutzern ermöglichen, zu kooperieren, zu kommunizieren, sich abzustimmen und zu handeln [PBS+00]. Das Ziel des DREAM-Projekt ist es, eine vollständige, robuste und skalierbare Entwicklungs- und Laufzeitumgebung für evolutionäre Berechnungen im Internet bereitzustellen.

Als Teil dieses Projektes wurde in [JPP02] ein Werkzeug entwickelt, um verteilte Experimente im Internet ausführen zu können. Obwohl sich das Projekt hauptsächlich mit evolutionären Algorithmen beschäftigt, unterstützt die Umgebung jegliche Anwendung, die massiv parallelisierbar, asynchron, ressourcenhungrig und robust ist, solange das Nachrichtenvolumen zwischen den einzelnen Teilprozessen gering ist. Um auch Weitverkehrsverbindungen unterstützen zu können, in denen die Kommunikationskosten deutlich höher liegen als in lokalen Netzen, wird auch in DREAM auf das P2P-Paradigma gesetzt. In [JPP02] beschreiben Jelasity et al. primär die zu Grunde liegende Netzwerkschnittstelle, die nicht mehr auf evolutionäre Algorithmen fixiert ist, die sog. *Distributed Resource Machine (DRM)*. Hier wird eine lokale Sicht auf eine Menge von Prozessen gebildet, welche über das gesamte Netz verteilt ablaufen.

Die DRM-Knoten bilden ein selbstorganisierendes Netzwerk. Durch regelmäßigen Austausch und Abgleich ihrer Nachbarschaftslisten mit aus ihrer Liste zufällig gewählten Knoten (siehe auch den Abschnitt 2.5 über epidemische Algorithmen) wird im Netz verbreitet, welche Knoten unter welchen Adressen verfügbar sind. Um die Skalierbarkeit zu gewährleisten, ist die Länge dieser Nachbarschaftslisten begrenzt. Die Robustheit des DRM-Ansatzes wird mit dem zugrunde liegenden epidemischen P2P-Ansatz begründet. Ein Ausfall selbst einer Vielzahl von Knoten macht das Netz nicht unbrauchbar. Dazu wurden in [JPP02] auch empirische Experimente durchgeführt, bei denen die Hälfte aller aktiven Knoten plötzlich entfernt wurde, ohne die Stabilität signifikant zu beeinflussen.

Nach [ACE+02] bietet das gesamte DREAM-Framework mehrere Einstiegspunkte für verschiedene Nutzer, angefangen bei einer abstrakten und einfachen grafischen Schnittstelle bis hin zu einer Schnittstelle, die direkt auf der DRM aufsetzt.

OrganicGrid Chakravarti et al. [CBL04] stellen OrganicGrid vor. OrganicGrid verfolgt ebenso einen P2P-basierten Ansatz. Aufträge werden nicht zentral gestartet, in Teilaufträge zerlegt und von dort aus gezielt über einen zentralen Scheduler verteilt, sondern sie werden in Form von Agenten in das P2P-Netz gebracht, die sich dort teilen und verbreiten können.

Bei der Verteilung wird dabei ein Baum aufgebaut, an dem entlang auch Ergebnisse zurückgereicht werden sollen. Aufträge werden beim OrganicGrid nicht aktiv verteilt, sondern die Knoten fragen bei Ihren Nachbarn regelmäßig an, ob diese Arbeit für sie haben. Dies verursacht auch im unbelasteten Fall (d.h. wenn keine Arbeit vorliegt) eine gewisse Mindestlast.

Robustheit gegenüber Knotenausfällen wird bei OrganicGrid dadurch erreicht, dass so lange Aufträge vergeben werden, bis alle Teilergebnisse zurück sind. Dies führt zu einer gewissen Redundanz, da gerade in einem unbelasteten Netz ein Teilauftrag im worst-case von allen Knoten gleichzeitig berechnet wird.

CompuP2P Ein Ansatz, der auch eine Bezahlung von erbrachter Leistung vorsieht, ist CompuP2P [GS04]. Auch CompuP2P basiert auf P2P-Netzen und benutzt Ideen der Spieltheorie und aus der Mikroökonomie. Bevor ein Auftrag von einem Knoten an einen anderen geschickt wird, wird zwischen diesen beiden Knoten selbständig (d.h. ohne zentrale Kontrolle) ein Preis für diese Berechnung ausgehandelt.

Die Adressierung bei CompuP2P basiert auf dem Chord-Protokoll [SMLN+03], soll aber mit geringen Änderungen austauschbar durch andere strukturierte oder unstrukturierte Protokolle sein. Einzelne Knoten werden hier nicht als bösartig, aber als eigennützig betrachtet, d.h. die Knoten streben danach, ihren Gewinn zu maximieren.

Eine gewisse Form von Ausfallsicherheit soll dadurch gewährleistet werden, dass von Zeit zu Zeit Checkpoints der Berechnung gemacht werden. Da auf einen zentralen

Checkpoint-Server aus Gründen der Ausfallsicherheit und Skalierbarkeit verzichtet werden soll, wird auch dieser Dienst aus dem P2P-Netz erbracht, durch Bereitstellung von Speicherplatz – gegen Bezahlung natürlich.

JXTA JXTA™ [JXT] wurde ursprünglich von Sun Microsystems™ entwickelt, wird aber mittlerweile als OpenSource-Projekt geführt. JXTA ist eine Sammlung von Protokollen, die es Geräten mit Netzwerkverbindung (vom Handy über PDAs bis hin zu Servern) ermöglicht, in einem P2P-Netz zu kommunizieren und miteinander zu arbeiten. JXTA-Peers bauen ein virtuelles Netzwerk auf, in dem jeder mit jedem interagieren kann und direkt die Ressourcen des anderen nutzen kann, unabhängig davon, ob einer der beteiligten Knoten durch eine Firewall abgeschirmt wird oder auf der Netzwerkschicht unterschiedliche Basisprotokolle verwendet werden.

Nach [Gon01] ist die abstrakte Repräsentation eines Knotens ein *Peer*-Objekt. Peers können zu Gruppen zusammengefasst werden, welche über entsprechende Protokolle gefunden werden können. Nachrichten werden in JXTA über sogenannte *Pipes* ausgetauscht. Diese Pipes sind unidirektional, können aber an beliebige Peers gebunden werden. Dies bedeutet, dass mehrere Ein- und Ausgänge möglich sind. Es gibt aber auch spezielle Punkt-zu-Punkt-Pipes, die nur je einen Ein- und Ausgang erlauben. Der Nachrichtenaustausch und auch die Konfiguration in JXTA erfolgt vollständig in XML.

Verbeke et al. [VNRS] präsentieren ein auf JXTA-basierendes Framework für Berechnungen in einer heterogenen dezentralen Umgebung. Auch dieser Ansatz leistet einen Dienst für die parallele Berechnung von Teilproblemen, es ist keine Kommunikation oder Kooperation zwischen den rechnenden Instanzen vorgesehen. Jeder teilnehmende Knoten kann verschiedene Rollen einnehmen und damit auch gleichzeitig Teil einer Gruppe von Peers der gleichen Rolle sein. Diese definierten Rollen sind z.B. *Worker*, *Monitor*, *Dispatcher* oder *Repository*.

Beim Verteilen eines Auftrags wendet sich ein potentieller Auftraggeber an die oberste Monitor-Gruppe, welche seine Anfrage an einen zuständigen Task-Dispatcher weiterleitet, der den Auftrag entgegennimmt und diesen dann an die Worker in seiner Gruppe verteilt. Die Repositories dienen dazu, Code und Teilaufgaben zu speichern, bis sie von einem Worker heruntergeladen werden. Auch die Ergebnisse werden im Repository abgelegt und dort vom Initiator wieder abgerufen. Der Initiator erfährt vom Ende der Berechnung dadurch, dass er regelmäßig beim zugehörigen Task Dispatcher anfragt, ob der Auftrag fertig berechnet ist.

Ausfallsicherheit wird dadurch gewährleistet, dass die Kommunikation immer mit einer Gruppe und nicht mit einzelnen Knoten erfolgt. Dadurch, dass Knoten mit besonderen Aufgaben ständig dafür sorgen, dass sie nicht alleine in der Gruppe sind, die diese Aufgaben erfüllen soll und innerhalb dieser Gruppen Zustände repliziert werden, kann bei Ausfall eines Knotens ein anderes Gruppenmitglied die Aufgabe übernehmen. Der ausgefallene Knoten kann dann dadurch ersetzt werden, dass ein anderer Knoten in die Gruppe aufgenommen wird und an der Zustandsreplikation teilnimmt.

Skalierbarkeit soll durch die Einführung einer Hierarchie gewährleistet werden. Die Last soll dabei auf viele Knoten verteilt werden. Ein Problem dabei ist allerdings, dass eingehende Nachrichten zur Jobverteilung oder auch einfach der Beitritt zum Netzwerk immer über die oberste Monitor-Gruppe laufen muss. Jeder Teilnehmer dieser Gruppe erhält die Nachricht. Durch vorher abgestimmte Zuständigkeiten (z.B. durch dynamisches Scheduling) muss jedoch nur ein Knoten diese Nachricht bearbeiten. Allerdings muss jeder der Knoten bis zu diesem Entscheidungspunkt die eingehende Nachricht verarbeiten. Bei starker Last kann dies zum Flaschenhals werden.

G2:P2P Bei G2:P2P [MK03] wird im Rahmen des G2-Projektes versucht, mittels des Remoting-Mechanismus von Microsoft .NET Aufträge im Netz zu verteilen. Das Remoting-Framework von Microsoft .NET kann in etwa mit Java RMI verglichen werden.

Um die teilnehmenden Knoten zu verwalten wird mittels Pastry [RD01] ein strukturiertes Netz aufgebaut. Innerhalb dieses strukturierten Netzes werden dann über einen Remoting-Kanal auf entfernten Knoten Prozesse gestartet und über den gleichen Kanal die Ergebnisse eingesammelt. Die Methode zum Starten des entfernten Prozesses terminiert sofort. Ergebnisse müssen nach dem Ende der Berechnung durch den Initiator beim entfernten Knoten abgerufen werden.

Die Tatsache, dass die Berechnung beim entfernten Knoten beendet ist, kann auf zwei Arten festgestellt werden: Zum Einen über eine Callback-Benachrichtigung durch den entfernten Knoten, zum Anderen durch regelmäßiges Anfragen beim entfernten Knoten *(Polling)*.

Peer Service Networks Bei *Peer Service Networks – Distributed P2P Middleware* [HB03] von Harwood und Balsys wird zwar Wert auf eine Abstraktion der Netzwerk-Adressen von zur Verfügung stehenden Peers gelegt – der Nutzer sieht einfach eine Menge an Knoten, die ihm für Berechnungen zur Verfügung stehen – aber dort steht ihm nur eine Art „verteilte Shell“ zur Verfügung. Durch den verbindungsorientierten Ansatz dieser Arbeit kann man nicht einmal unbedingt von P2P sprechen, da der inhärent dynamische Charakter von P2P-Netzen hier nicht berücksichtigt wird.

Das Hauptaugenmerk wurde bei dieser Arbeit auf Sicherheitsaspekte gelegt. Die verwendeten Kanäle zwischen den Knoten sind verschlüsselt und Nutzer müssen sich anmelden und authentifizieren, um den Dienst nutzen zu können.

2.3. Client-Server-Systeme

Das klassische Client-Server-Paradigma gibt eine strikte Rollentrennung zwischen den Systemen vor. Einer der beiden Kommunikationspartner übernimmt die Rolle des Servers, der andere übernimmt die Rolle des Clients.

Das Client-Server-Paradigma ist weit verbreitet. So basiert der meistgenutzte Dienst des Internet, das WorldWideWeb (WWW) auf dem HTTP-Protokoll [FGM+99]. So lassen sich Rechner in der Rolle eines Clients Webseiten von einem Server liefern. Um mit den Anfragen vieler Clients gleichzeitig klar zu kommen, werden Server auch häufig mehrfach redundant ausgelegt, so dass diese Anfragen parallel beantwortet können. Hier erkennt man eine große Schwachstelle an den Client-Server-Systemen. Da ein Dienst nur an einer Stelle (vom Server) angeboten wird, muss mit steigender Anzahl der Dienst-Nutzer auch die Leistungsfähigkeit des Server gesteigert werden. Diese Fähigkeit bezeichnet man als Skalierungsfähigkeit, welche beim Client-Server-Paradigma nicht bzw. nur in gewissem Rahmen (durch größeren Hardwareeinsatz) gegeben ist. Der Server selber oder auch seine Netzwerkanbindung an das Internet ist ein Flaschenhals.

2.4. Peer-to-Peer-Netze

Unter einem P2P-Netz versteht man kurz gefasst ein Netz von unabhängigen Rechnern (Peers), welche über ein Kommunikationsmedium in beliebiger Form miteinander verbunden sind, wobei keiner der beteiligten Rechner eine ausgezeichnete Rolle einnimmt. Allgemein geht man hierbei davon aus, dass das zugrunde liegende Netz das weltweite Internet ist und damit in der Art und Weise der angeschlossenen Rechner nicht heterogener sein könnte.

Ein P2P-Netz ist ein sogenanntes Overlay-Netz, also ein Netz, das komplett von den Routing-Strukturen des zugrunde liegenden Netzes abstrahiert und Punkt-zu-Punkt-Verbindungen zwischen den beteiligten Knoten unterhält (siehe auch Abbildung 2.1). Diese Punkt-zu-Punkt-Verbindungen können strukturiert gewählt und über einen verteilten Kontrollalgorithmus ausgewählt sein, können aber ebenso rein zufällig gewählt werden. Wichtig ist nur, dass es keine zentrale Instanz gibt, die das Netz kontrolliert oder überwacht oder in sonst einer Art beeinflusst. Solch eine Instanz behindert immer nur die Skalierbarkeit (s.u.).

Ein Charakteristikum von P2P-Netzen ist, dass dort prinzipiell ein ständiges Kommen und Gehen herrscht. Knoten können aus dem Netz herausfallen (sei es wegen absichtlicher Abschaltung oder wegen eines Verlustes der zugrunde liegenden Netzwerkverbindung). Genauso können laufend Knoten hinzukommen, sei es ein „echt neuer“ Knoten oder einfach ein Knoten, der nach einer Unterbrechung wieder in das Netz zurückkehrt.

2.4.1. Vorteile von P2P-Netzen

P2P-Netze bieten gegenüber dem Client-Server-Ansatz einige Vorteile:

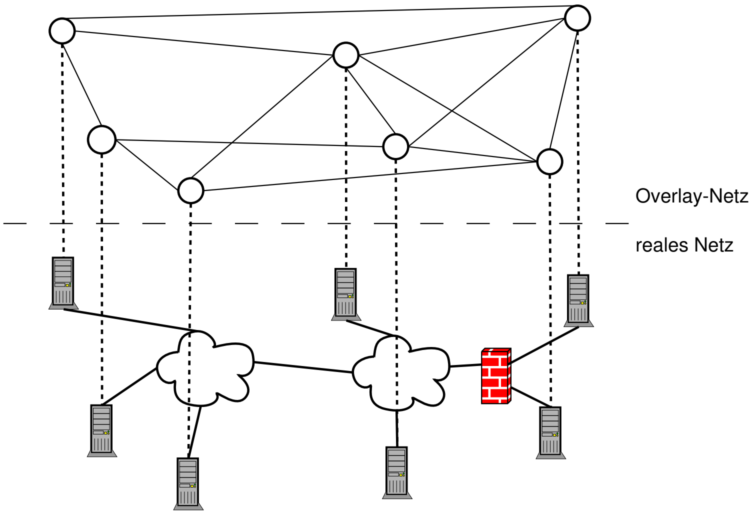

Abbildung 2.1.: Overlay-Netz

Skalierbarkeit

Durch die Gleichberechtigung aller Teilnehmer wird die Last gleichmäßig auf alle Teilnehmer verteilt. Es müssen natürlich verteilte Algorithmen eingesetzt werden, die mit der Anzahl der Teilnehmer auch skalieren. So hatten anfängliche P2P-Protokolle das Problem, dass z.B. Suchnachrichten mittels Broadcast verschickt wurden. Bei n Knoten verursacht dies aber $O(n)$ Nachrichten, was schnell das Netzwerk überlastet. Durch das gleichzeitige Suchen mehrerer Knoten waren dann sehr viele Nachrichten zur gleichen Zeit unterwegs. Durch das strukturierte Ablegen von Informationen können diese aber auch effizienter gefunden werden. Mit aktuellen Algorithmen, wie z.B. Chord [SMLN$^+$03], Pastry [RD01], Tapestry [ZKJ01], OceanStore [KBC$^+$00] oder auch Abwandlungen derer [MG05] können Informationen mit geringem Nachrichtenaufwand gefunden und Informationen schnell verbreitet werden, i.A. mit Zeitaufwand im Bereich $O(log(n))$.

Robustheit und Verlässlichkeit

Durch den Verzicht auf zentrale Komponenten wird der P2P-Ansatz sehr robust. Wenn ein Knoten ausfällt, übernimmt ein anderer automatisch seine Aufgabe. In P2P-Netzen wird für die Erbringung eines bestimmten Dienstes zu keinem Zeitpunkt das Vorhandensein eines bestimmten Knotens vorausgesetzt, das heißt aber auch, dass bei Bedarf der Zustand dieser Komponente repliziert werden muss, damit der Dienst ausfallsicher ist. Wenn das Gesamtsystem von einer einzigen zentralen Komponente abhängig ist, kann ein Ausfall dieser Komponente das gesamte System lahmlegen. Da es bei echten

P2P-Netzen diese zentrale Komponente per Definition nicht gibt, besteht dort dieses Risiko nicht.

Wartungsaufwand

Dadurch, dass P2P-Netze sich selbst organisieren und sich automatisch eine Topologie ergibt, ist keinerlei administrativer Aufwand nötig, wenn neue Knoten hinzukommen oder Knoten das Netz verlassen. Dies ist ein großer Unterschied zu Cluster-Systemen, wo Aufgaben zwar auch massiv parallel gerechnet werden, jedoch jeder Knoten einzeln im Netz bekannt gemacht und konfiguriert werden muss. Dies erfolgt bei P2P-Netzen ganz von alleine. Ein Knoten, der neu hinzukommt, wird auch annähernd sofort sichtbar und nutzbar, womit sich die Dynamik und Flexibilität des Netzes erheblich erhöht. Die Zeitdauer, bis ein neuer Knoten im Netz bekannt ist, hängt maßgeblich von den verwendeten Algorithmen ab.

P2P-Netze wurden anfangs hauptsächlich für urheberschutzrechtlich bedenkliche Musik- und Film-Tauschbörsen [HSS02] eingesetzt. Mittlerweile werden zunehmend Anstrengungen unternommen, das Gebiet der P2P-Systeme für rechtlich unkritische Technologien zu nutzen.

Zu solchen Technologien zählen auch Dienste, die CPU-Leistung, die auf einem Großteil der am Internet (permanent oder zeitweise) angeschlossenen Rechner brach liegt, für verteilte Berechnungen zu nutzen. Die bereits vorgestellten Ansätze wie SETI@home, Folding@home, etc. sind allerdings nicht als echte P2P-Anwendungen zu zählen, da bei diesen die beteiligten Knoten niemals untereinander kommunizieren, sondern immer nur mit dem zentralen Server interagieren.

2.4.2. Nachteile von P2P-Netzen

P2P-Systeme sind jedoch nicht das Allheilmittel, sie haben auch Nachteile gegenüber Client-Server-Systemen:

Globales Wissen

In zentralistischen Systemen lässt sich leicht ein Überblick über das Gesamtsystem erstellen. Dieser Überblick beinhaltet z.B. Statistiken oder Abrechnungsdaten. In verteilten Systemen ist es schwer, eine konsistente Sicht auf diese Daten zu haben. Die für eine Statistik relevanten Informationen müssen verteilt im Netz abgelegt werden und dann bei Bedarf von dort eingesammelt werden. Eine Konsistenzgarantie dafür ist mit großem Aufwand verbunden.

Sicherheitskritische Daten

Viele Systeme verwalten sicherheitskritische Daten (z.B. Passwörter, Adressen). Bei der Handhabung solcher Daten ist man bestrebt, diese an so wenigen Stellen wie möglich abzulegen, damit so wenig Orte wie möglich entstehen, an denen jemand die Daten einsehen kann. Üblicherweise werden sie in einer zentralen Datenbank verwaltet. Um die Ausfallsicherheit und Unabhängigkeit der P2P-Systeme von zentralen Diensten zu gewährleisten, müssten die sicherheitskritischen Daten aber über mehrere Rechner verteilt werden. Ein Schutz der Daten muss aufwändig realisiert werden.

2.5. Epidemische Algorithmen

Der hier vorgestellte Ansatz für epidemische Algorithmen für verteilte Anwendungen basiert auf den Ausführungen von Eugster et al. [EGKM04]. Dort werden auch mathematische Modelle für die nachfolgenden Ausführungen geliefert.

Epidemische Algorithmen – angelehnt an biologische Epidemien – haben sich in letzter Zeit als robuste und skalierbare Möglichkeit zur Nachrichtenausbreitung in verteilten Systemen einen Namen gemacht. Ein Prozess, der eine Information im System verbreiten möchte, schickt diese nicht an einen zentralen Server oder eine Server-Farm, wo diese dann weiter verteilt wird, sondern einer zufällig gewählten Menge von Peers. Jeder der Empfänger macht mit der Nachricht dasselbe, d.h. er verteilt sie weiter an eine zufällig gewählte Menge von Knoten. Das zugrunde liegende Prinzip ähnelt der Ausbreitung von Epidemien.

Epidemien sind sehr widerstandsfähig gegen Auslöschung. Dies macht sie sehr resistent gegen Fehler bei der Ausbreitung von Informationen. Selbst wenn mehrere Knoten während der Übertragung ausfallen, wird die Nachricht mit hoher Wahrscheinlichkeit an alle aktiven Knoten propagiert.

Die epidemische Ausbreitung einer Nachricht wird in Abbildung 2.2 gezeigt. Die Kanten entsprechen unidirektionalen Kommunikationskanälen. Beim angegebenen Beispiel werden neun von zehn Knoten innerhalb von drei Zyklen informiert. Der letzte Knoten wird nur deswegen nicht informiert, da ihn zu diesem Zeitpunkt kein anderer Knoten kennt. Sobald er selber jedoch eine Nachricht an einen anderen Knoten verschickt wird ihn dieser in seine Liste aufnehmen.

Algorithmen zur epidemischen Nachrichtenverbreitung sind relativ einfach strukturiert und einfach einzusetzen. Zusätzlich zu ihrer Skalierbarkeit, welche dadurch zustande kommt, dass die eigentliche „Arbeit“ bei der Verteilung, nämlich das Versenden der Nachrichten, auf alle Knoten des Netzes verteilt wird, sind epidemische Algorithmen sehr stabil, auch unter Berücksichtigung von Knoten- und Verbindungsausfällen. Es gibt keinen „single point of failure“ und die Verlässlichkeit des Dienstes nimmt nur langsam ab mit der Anzahl der ausgefallenen Knoten.

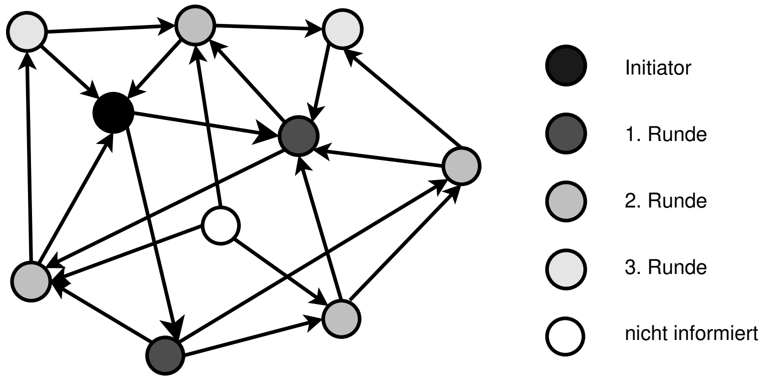

Abbildung 2.2.: Epidemische Nachrichtenausbreitung

In einem epidemischen Algorithmus nimmt jeder am Netzwerk beteiligte Knoten an der Nachrichtenausbreitung teil. Jeder Prozess puffert eingehende Nachrichten und schickt diese an eine bestimmte Zahl von Nachbarn weiter. In dem von Eugster et al. vorgestellten Ansatz gibt es drei Parameter, mit denen die Ausbreitung beeinflusst wird. Dazu gehören die Größe des Eingangspuffers, welche die Anzahl an Nachrichten begrenzt, die gleichzeitig verbreitet werden können, die Anzahl der Runden, in denen eine Nachricht verbreitet werden soll und die Anzahl der Knoten, die pro Runde eine Nachricht geschickt bekommen.

Es existieren viele Varianten für epidemische Nachrichtenverbreitung, die im Prinzip auf unterschiedlichen Werten dieser drei Parameter beruhen. Bei der Wahl der Parameter spielt eine Rolle, wie schnell sich eine Epidemie ausbreiten soll und wie stark ein Knoten mit der Nachrichtenausbreitung belastet werden soll. Prinzipiell müssen die Parameter auch nicht statisch gewählt werden, sie können auch dynamisch aufgrund des momentanen Systemzustandes gewählt werden.

Ein Teilproblem der epidemischen Algorithmen nach Eugster ist das der „Mitgliedschaft“ (*Membership-Problem*) in einem epidemischen Netz. Ein Knoten kann eine Nachricht natürlich nur an Knoten versenden, die er auch kennt. Es ist relevant, wie ein Knoten diese Nachbarschaftsliste aufbaut. Bei einem Einsatz in beliebig großen Netzen ist klar, dass ein Knoten keine vollständige Liste aller teilnehmenden Knoten vorhalten kann. Zum Einen wäre so ein Ansatz nicht skalierbar, weil mit wachsender Größe des Netzes der Speicherbedarf für die Nachbarschaftsliste linear wachsen würde. Zum Anderen ist die Dynamik des Netzes gerade bei der Anwendung der epidemischen Algorithmen in P2P-Netzen zu groß; der Aufwand für die Einhaltung einer Konsistenz der Nachbarschaftslisten wäre zu groß. So kommt man dazu, dass jeder Knoten nur eine Teilsicht auf das Netzwerk erhält. Hierbei muss zwischen Skalierbarkeit und Verlässlichkeit abgewogen werden. Große Listen fördern die Verlässlichkeit und stören die Skalierbarkeit, bei kleinen Listen verhält es sich entgegengesetzt.

Ein Ansatz für die Verwaltung der Mitgliedschaftslisten besteht darin, dass die Verwaltung gemeinsam mit der Nachrichtenausbreitung durchgeführt wird. Bei jedem

Versenden einer Nachricht wird die Nachbarschaftsliste des sendenden Knotens mitgeschickt (*piggy-backing*).

Ein ähnlicher Ansatz wird von Voulgaris et al. [VJvS03] beschrieben. Hier werden regelmäßig Nachrichten zwischen Knoten ausgetauscht, die Nachbarschaftslisten enthalten. Die aktuellsten Einträge werden beibehalten. Dieser Austausch findet parallel zur „normalen" Nachrichtenausbreitung statt. Hierbei existiert zwar im Vergleich zum ersten Ansatz eine höhere Netzwerklast, jedoch bleibt dieser Faktor konstant und ist nicht von der Größe des Netzes abhängig. Im Ausgleich dafür findet die Propagierung von Änderungen der Teilnehmerlisten deutlich schneller statt und ist unabhängig von der Anzahl der sich ausbreitenden Informationen im Netz.

2.6. IP-Multicast

Eine effiziente Verteilung von Nachrichten kann auch innerhalb der Basisprotokolle auf der Netzwerkschicht, also im Rahmen des Internet innerhalb des IP-Protokolls erfolgen. Statt Nachrichten an jeden Empfänger einzeln zu verschicken, wird eine Nachricht an eine sog. Multicast-Gruppe versandt und die Netzwerkschicht kümmert sich darum, die Nachricht an alle Empfänger zu übermitteln.

Die Bildung der Multicast-Gruppen erfolgt dabei über das IGMP-Protokoll [Fen97]. Hierzu teilt jeder Knoten über dieses Protokoll seinem nächsten Router mit, dass er Teilnehmer dieser Gruppe ist. Der Router verwaltet entsprechende Mitgliedschaftstabellen und leitet die Mitgliedschaft an seine benachbarten Router weiter, sofern er nicht schon eine Mitgliedschaft für einen Knoten in seinem Bereich registriert hat. Für den entstehenden Verteilungsbaum gibt es primär zwei Varianten:

SourceTree: Hierbei hält jeder Router Routingtabelleneinträge für Paare von Quelladresse und Multicast-Zieladresse vor. So entstehen optimal kurze Wege von der Quelle zu jedem einzelnen Zielhost (siehe Abbildung 2.3).

SharedTree: Bei diesem Verfahren wird für jede Multicast-Gruppe ein gemeinsamer Wurzelknoten des Verteilungsbaumes gewählt, genannt *rendezvous point*. Von diesem Knoten aus werden Pakete dann an die eigentlichen Empfänger verschickt, die Router müssen nur einen Routingeintrag für jede Multicast-Adresse verwalten. Der Speicherbedarf und wegen kleinerer Tabellen auch die lookup-Zeit sind bei diesem Verfahren deutlich geringer, jedoch sind die entstehenden Wege und damit Latenzzeiten länger (siehe Abbildung 2.4).

IP-Multicasting ist durch die Unterstützung in der Netzwerkschicht und die Minimierung der Nachrichtenanzahl deutlich effizienter im Hinblick auf die Nachrichtenkomplexität. Allerdings muss hier vorausgesetzt werden, dass die Komponenten der Netzwerkschicht zwischen Sender und Empfänger, also alle dazwischen liegenden Internet-

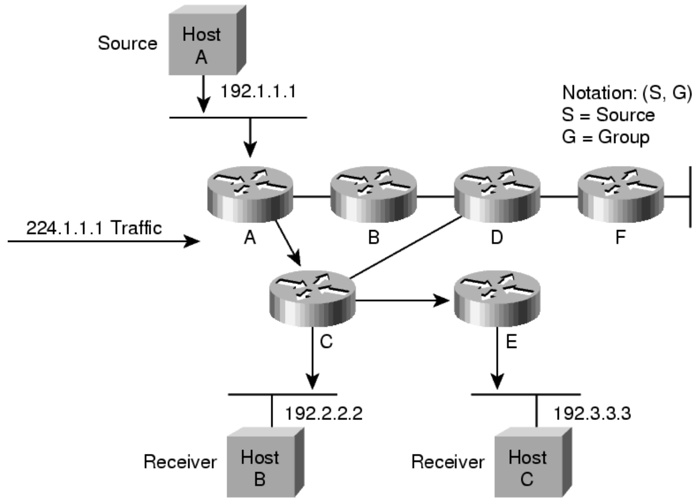

Abbildung 2.3.: Multicast: SourceTree

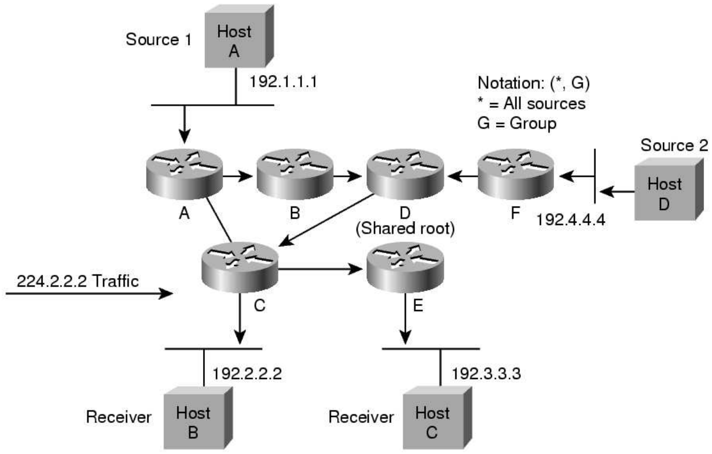

Abbildung 2.4.: Multicast: SharedTree

Router, das IGMP-Protokoll sprechen und potentiell beeinflussen können. Ein weiteres Problem stellt der Gebrauch von Firewalls dar. Wenn auf dem Weg eine Firewall entweder keine Multicast-Pakete oder auch die Kontrollpakete für die Gruppenmitgliedschaft (IGMP) nicht durchlässt, so sind die dahinter liegenden Knoten nicht erreichbar.

Eine Lösung für das Problem, dass alle Router auf dem Weg von der Quelle zum Ziel IGMP-fähig sein müssen, besteht darin, dass IGMP-fähige Router Unicast-Punkt-zu-Punkt-Verbindungen aufbauen und damit ein Multicast-Overlay-Netz konstruieren, den sogenannten MBONE.

Das IGMP-Protokoll garantiert in keiner Form, wie lange es dauert, bis alle teilnehmenden Router ein Update der Mitgliedschaftsdaten verarbeitet haben, d.h. bis die Teilnehmerliste wieder stabil und konsistent sind. Da Multicast-Umgebungen aber sehr dynamisch sein können, können potentiell ständig Knoten hinzukommen und wieder wegfallen.

Ein weiterer Aspekt von IP-Multicast ist, dass dieser unzuverlässig ist, d.h. dass in keiner Form garantiert wird, dass alle Gruppenteilnehmer auch tatsächlich jede an die Gruppe gesendete Nachricht erhalten. Dazu existieren jedoch Erweiterungen bzw. eigene Protokolle, die im nächsten Abschnitt vorgestellt werden.

Verlässlicher Multicast

Basierend auf dem ursprünglichen IP-Multicast nach [Fen97] wurden einige Protokolle entwickelt, die den Multicast verlässlich machen sollten.

Einige Ansätze sollen kurz vorgestellt werden:

Multicast Transport Protocol (MTP) [AFM92]: MTP ist ein sehr frühes verlässliches Multicast-Protokoll, welches auch schon als RFC festgeschrieben ist. Ein Knoten kann in diesem Netz eine von drei Rollen einnehmen, als *Master*, als *Sender/Empfänger* oder als reiner *Empfänger*. Der Master kontrolliert dabei die Liste der Gruppenteilnehmer. Um die hohe Performance von lokalen Netzen auszuschöpfen, basiert MTP auf Negativ-Quittungen, d.h. dass im Nicht-Fehlerfall keine zusätzliche Last durch Quittungen erzeugt wird.

Reliable Multicast Transport Protocol (RMTP) [LP96]: RMTP bietet einen Dienst, der Nachrichtenpakete von einem Sender zu mehreren Empfängern („one-to-many“) in korrekter Reihenfolge liefert. Der Ansatz basiert, um Skalierbarkeit zu gewährleisten, auf einem hierarchischen System, das ursprünglich bei Paul et al. [PSK94] vorgestellt wurde. Die Knoten auf dem Weg von Sender zu Empfänger fungieren als Cache, um den Aufwand der Neuübertragungen möglichst klein zu halten.

Illinois Reliable Multicast Architecture (IRMA) [LHB99]: IRMA stellt einen zuverlässigen Dienst bereit, der Pakete in korrekter Reihenfolge ACK-basiert von einer

Quelle an alle Empfänger der Gruppe zustellt. Im Gegensatz zu vielen anderen Diensten wird dafür TCP benutzt, obwohl der bei TCP verwendete Verbindungsaufbau nicht Multicast-geeignet ist, da eine eins-zu-eins Abstimmung über Sequenznummern erfolgt. Dieses Problem wird bei IRMA mit entsprechenden IRMA-fähigen Routern gelöst, was diese Lösung schwer einführbar macht.

Pragmatic General Multicast (PGM) [GMSC03]: PGM ist ein verlässliches Multicast-Transport-Protokoll für Anwendungen, die Daten von einer Quelle zu vielen Zielen transportieren. Als Basis benutzt PGM den IP-Multicast. PGM garantiert, dass ein Empfänger in der Gruppe entweder alle Pakete der Übertragung in der richtigen Reihenfolge erhält, oder aber das ihm ein Fehler angezeigt wird, der signalisiert, dass ein nicht behebbarer Netzwerkfehler aufgetreten ist. Skalierbarkeit wird durch die Nutzung von Hierarchieebenen, durch Unterdrückung von Negativ-Quittungen und durch eine Vorwärts-Fehlerkorrektur realisiert.

Eine Klassifikation verschiedener Ansätze für verlässlichen Multicast findet sich bei Atwood [Atw04]. Vergleiche und Übersichten über verschiedene Multicast-Protokolle haben auch Whetten et al. [WT00] und Levine et al. [LJLA98] angestellt.

Untersuchungen über die Skalierbarkeit von verlässlichen Multicast-Protokollen führten Kasera et al. [KHTK00] auf Basis mehrerer Multicast-Kanäle, Barcellos et al. [BE98] mittels Polling und Yoon et al. [YLY+02] mittels eines kombinierten Gruppen/Baum-Ansatzes durch.

Abschließend soll noch mit konkretem Bezug auf diese Arbeit auf eine aktuelle Publikation von Radoslavov et al. [RPGE04] hingewiesen werden, die einen Vergleich zwischen anwendungsbasierten und Router-unterstützen hierarchischen Ansätzen für verlässlichen Multicast zum Inhalt hatte. Ursprünglich sollte hier gezeigt werden, dass der Router-unterstützte Ansatz deutlich bessere Performance zeigt als der anwendungsbasierte Ansatz. Jedoch hat sich im Laufe der Arbeit herausgestellt, dass sogar ein relativ einfacher anwendungsbasierter Algorithmus vergleichbar gut funktioniert.

Der Vollständigkeit halber soll noch auf den Ansatz von Eugster und Guerraoui [EG02] hingewiesen werden. Dort wird versucht, den Multicast mittels epidemischer Algorithmen effizient zu lösen.

3. Systementwurf

In diesem Kapitel wird dargestellt, aus welchen Komponenten eine Middleware zur verteilten Berechnung aufgebaut sein kann und wie konzeptionell die Verteilung von Arbeitsaufträgen im Netz erfolgen kann. Hierbei wird Wert auf ein modulares und austauschbares Modell gelegt, erst im folgenden Kapitel wird dann eine konkrete Implementierung des hier vorgestellten Entwurfs präsentiert.

3.1. Arten von Berechnungen

Damit eine Berechnung verteilt erfolgen kann, muss der Auftrag sich in einzelne parallel lauffähige Teile zerlegen lassen. Grob kann man dabei die verteilten Berechnungen in zwei verschiedenen Klassen einteilen.

- Der Auftrag kann man in mehrere identische Arbeitsaufträge geteilt werden, dort ist nur wichtig, n verschiedene Resultate berechnet zu haben (z.B. durch zufallsbasierte Algorithmen)
- Jeder Teilauftrag bekommt unterschiedliche Parameter mit, zu jedem Satz Parameter muss der zugehörige Satz Ergebnisse vorliegen, damit die Berechnung abgeschlossen ist (z.B. Fraktalberechnung).

Bei beiden Klassen der Berechnung wird zusätzlich danach unterschieden, ob die Teilaufträge kooperieren, d.h. ob die die Knoten von der Existenz anderer Knoten mit der gleichen Aufgabe wissen und Nachrichten austauschen (enge Kopplung). Bei zahlreichen Ansätzen zur verteilten Berechnung ist dieser Aspekt nicht berücksichtigt. Hier werden Teilaufträge parallel auf verschiedenen Maschinen, aber vollständig unabhängig voneinander berechnet (lose Kopplung).

3.2. Jobverteilung

Ein wichtiger Aspekt der Middleware ist es, die unterschiedlichen Teilaufträge möglichst effizient und möglichst schnell zu verteilen. Hierzu wird jeweils die Beschreibung einer Teilmenge der Teilaufträge an andere Knoten verschickt, welche dann aus der Beschreibung entnehmen können, wo der Anwendungscode und eventuell vorhandene

Parameter heruntergeladen werden können und wohin die Ergebnisse geschickt werden sollen. Hierbei bleibt es der konkreten Implementierung der P2P-Ebene überlassen, ob der Wurzelknoten (der „Auftraggeber“) alle Arbeitsaufträge einzeln an andere Knoten verschickt, oder ob er jeweils ganze Aufgabenpakete an andere Knoten sendet, welche dort wieder in kleinere Einheiten zerlegt und weiterverschickt werden.

Ein wichtiger Grundsatz ist auf jeden Fall, dass ein Knoten sich nicht darauf verlassen darf, dass (a) andere Knoten existieren, die mitrechnen und (b) ein Knoten, der ein Auftragspaket akzeptiert, dieses auch tatsächlich fertig rechnet und das Ergebnis abliefert. Solche Probleme können durch Knotenausfälle, verlorene Nachrichten im Verbindungsnetzwerk oder auch einfach durch bewusste Entscheidungen in den Rechenknoten auftreten. Ein Knoten sollte sich also auf jeden Fall selber an seiner eigenen Berechnung beteiligen, damit die Berechnung durch mindestens einen Knoten sichergestellt ist.

3.3. Ein Chord-ähnlicher Algorithmus

In „*Efficient Broadcasts in P2P Grids*“ [MG05] wird eine Möglichkeit zum effizienten Nachrichtenbroadcast in P2P-Netzen beschrieben. Der Ansatz basiert darauf, dass dem unstrukturierten Netz eine Struktur aufgeprägt wird und aus der Sicht eines Knoten ein bestimmter Knoten für einen Teil der anderen Knoten die Verantwortung trägt, diesen eine neue Information mitzuteilen. Im konkreten Anwendungsfall dieser Arbeit bezieht sich die Information darauf, dass ein neuer Rechenauftrag zur Verfügung steht.

Ein optimaler Broadcast in einem beliebigen unbekannten zusammenhängenden Graphen besteht aus genau $2e - v + 1$ Nachrichten, sei e die Anzahl der Kanten und v die Anzahl der Knoten. Zur Optimierung kann der Graph in einen spannenden Baum transformiert werden, optimal sind dann $v - 1$ Nachrichten.

Ein naiver Broadcast-Algorithmus in einem P2P-Netz könnte sich alleine auf die epidemische Ausbreitung einer Nachricht verlassen. Dabei wird zu keinem Zeitpunkt garantiert, dass alle Knoten die Information erhalten. Mit zunehmender Zeit seit Ausbreitungsstart wird es immer unwahrscheinlicher, dass ein Knoten die Information noch nicht erhalten hat. Jedoch ist bei diesem Ansatz die Redundanz bzw. der Overhead sehr groß. Die Nachricht wird einem Knoten häufig mehrfach übermittelt.

Um diesen Overhead einzusparen, wird dem an sich unstrukturierten P2P-Netz eine Struktur aufgeprägt. Diese Struktur basiert darauf, dass jeder Knoten im P2P-Netz eine eindeutige Id trägt. Dabei kann man jeden Knoten auf einem Ring anordnen. Jetzt kann jeder Knoten eine Tabelle verwalten, die Zeiger auf bestimmte Positionen, d.h. aus seiner Sicht bestimmte Punkte im Ring (6 Uhr, 3 Uhr, 1 Uhr 30, ...) beinhaltet. Sollte gerade kein aktiver Knoten mit dieser idealen Id bekannt sein, so wird die Id des nächsten aktiven Knotens, der auf diesen Knoten folgt, gespeichert. Ein ähnlicher Ansatz wurde auch beim Chord-Protokoll [SMLN$^+$03] schon gewählt.

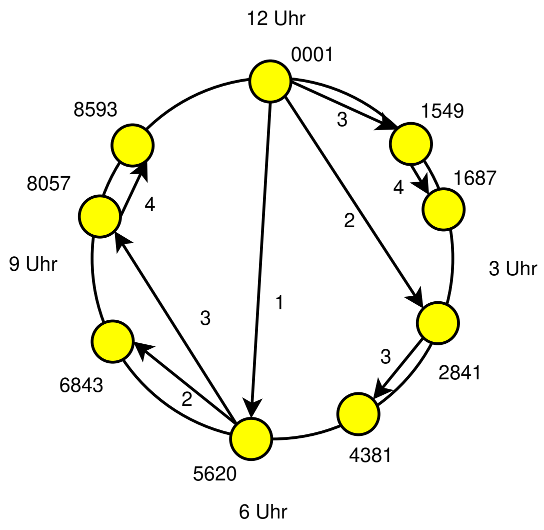

Abbildung 3.1.: Chord-ähnlicher Broadcast

Beim Verteilen von Informationen wird jetzt als erstes eine Nachricht an den im Ring gegenüberliegenden Knoten („auf 6 Uhr“) verschickt. Dieser Knoten erhält gleichzeitig den Auftrag, den „hinter ihm“ liegenden Teil des Adressraums zu informieren. Der lokale Knoten fährt fort, indem er als nächstes dem Knoten einen Viertelkreis von ihm entfernt („auf 3 Uhr“) eine Nachricht schickt. Dieser bekommt gleichzeitig den Auftrag, den Viertelkreis, für den er zuständig ist, zu informieren. Die so informierten Knoten fahren mit dem Algorithmus ihrerseits fort, so dass mit jeder versandten Nachricht ein Teil der Verantwortung auf andere Knoten delegiert wird.

Eine grafische Darstellung des Verteilungsprozesses findet sich in Abbildung 3.1. Der gegenüberliegende Knoten für den Initiator 0001 wäre idealerweise der Knoten mit der Id 5001. Da dieser jedoch nicht existiert, nimmt seine Aufgabe der im Ring folgende Knoten, also derjenige mit der Id 5620 wahr. Dieser übernimmt die Verantwortung für den Adressbereich 5620 - 0000.

Im Idealfall (d.h. wenn keine Nachricht durch Fehler im Netzwerk verloren gegangen ist) wird somit in $log(n)$ Schritten mit $n-1$ Nachrichten bei n Knoten jeder Knoten informiert. Dabei muss kein Knoten mehr als $log(n)$ Nachrichten verschicken.

3.4. Struktur einer Middleware für verteilte Berechnungen

Die Funktionseinheiten der geplanten Middleware können in mehrere Schichten zerlegt werden, so dass jede Schicht einzeln für sich betrachtet werden kann und höherliegende Schichten von den tiefer liegenden Implementierungsdetails abstrahieren können. Hierdurch erhält man einen modularen Aufbau, so dass die Module einzeln für sich ausgetauscht werden können.

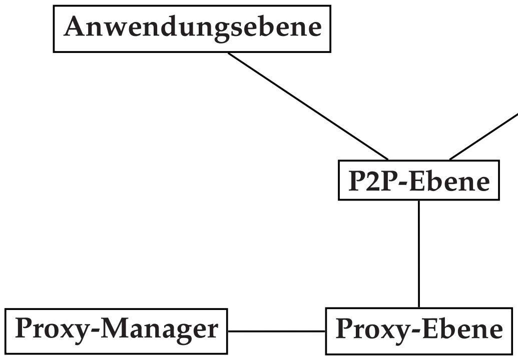

Abbildung 3.2.: Interface-Grobstruktur

Das vorgegebene Aufgabenspektrum kann in folgende Schichten bzw. Dienste zerlegt werden:

Client (User-Interface): Das User-Interface definiert Funktionen, die ein lokaler Benutzer auf der Middleware ausführen kann (Anmelden am Netz, Starten eigener Aufträgen).

P2P-Ebene: Diese Ebene stellt eine Menge von Funktionen zur Verfügung, mit denen auf ein P2P-Netzwerk zugegriffen werden kann. Es abstrahiert von der konkreten Implementierung eines P2P-Protokolls.

Proxy-Ebene: Die Proxy-Ebene definiert einen Nachrichtendienst, der auch hinter Firewalls funktioniert. Durch diese Ebene wird die transparente Nutzung eines Proxy-Servers zur Kommunikation ermöglicht.

Anwendungs-Ebene (Verteilte Anwendung): Diese Ebene bietet diejenigen Funktionen an, die einer verteilt laufenden Anwendung auf ihrer Laufzeitplattform zur Verfügung stehen.

Proxy-Manager: Der Proxy-Manager bietet Zugriff auf die Implementierung eines lokalen Proxy-Servers.

Die so eingeteilten Ebenen lassen sich hierarchisch wie in Abbildung 3.2 darstellen.

3.4.1. Callback-Struktur

Da zwischen zwei Schichten Nachrichten nicht nur nach „unten“ gegeben werden (d.h. Richtung Netzwerkschicht), sondern aufgrund der asynchronen Kommunikation auch

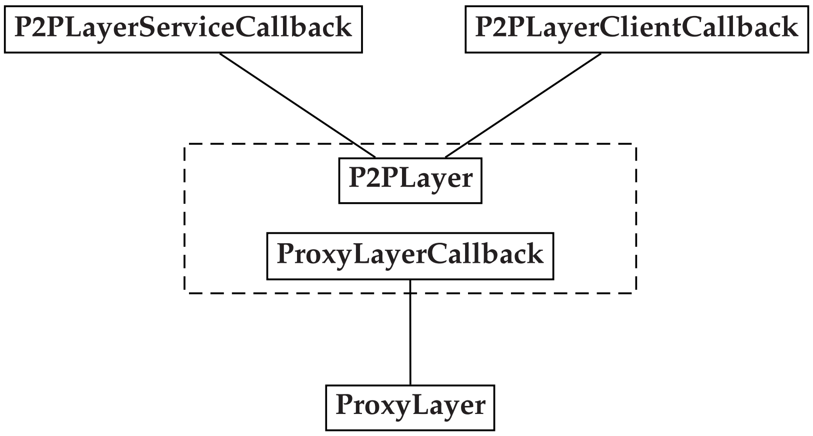

Abbildung 3.3.: Callback-Struktur

Nachrichten „zurück nach oben“ gereicht werden müssen, gibt es an jeder Schichtgrenze nicht nur ein Interface „nach unten“, sondern auch ein Callback-Interface „nach oben“. Instanzen der tiefer liegenden Schichten können über die Callback-Interfaces Nachrichten an die höherliegenden Schichten versenden.

Diese Implementierungsvariante ist de facto eine Realisierung des Observer-Patterns: Höherliegende Schichten implementieren ein Listener-Interface und registrieren sich bei den tiefer liegenden Schichten als Listener. Die Unterscheidung liegt aber eher in der Benennung der Schnittstellen und der Kommunikationspartner als im System. Beim Initialisieren muss das Callback-Interface bei der unteren Schicht registriert werden, welche dann die dort definierten Methoden nutzt. Das Pattern sieht nur eine Benachrichtigung über ein beliebiges Ereignis vor.

Eine Beispiel-Verknüpfung von Schichten mittels Callback-Interface kann man in Abbildung 3.3 am Beispiel der P2P-Schicht und der Proxy-Schicht sehen. *P2PLayer* und *ProxyLayerCallback* werden dabei von der gleichen Instanz implementiert.

3.4.2. Proxy-Ebene

Die Proxy-Ebene dient als unterste Schicht oberhalb der Java-Netzwerkschicht dazu, Basisnachrichten des Systems mittels TCP oder UDP an andere Teilnehmer des P2P-Netzes zu verschicken. Sie soll der darüber liegenden P2P-Schicht einen Nachrichtendienst zu Verfügung stellen, der davon abstrahiert, ob der Knoten hinter einer Firewall steht oder nicht, sowie allgemeine Basisfunktionen anbieten, die alle P2P-Protokolle benötigen. Ebenso könnte die Proxy-Schicht einen effizienten Multicast-Dient anbieten.

Die Proxy-Schicht leitet Nachrichten entweder an den zuständigen Proxy-Server weiter oder stellt Nachrichten direkt dem Empfänger zu. Für den Fall, dass der lokale Rechner nur über einen Proxy-Server mit dem P2P-Netz verbunden ist, werden eingehende Nachrichten über den TCP-Kanal vom Proxy-Server empfangen.

Für die wichtigsten Standardfunktionen wie z.B. „Verbindungsaufbau/-abbau zum P2P-Netz“ stehen Basisfunktionen bereit, für die auf der P2P-Ebene dann keine Nachrichtentypen mehr definiert werden müssen.

Über das zugehörige Callback-Interface können der P2P-Schicht eingehende Verbindungsaufbau-Versuche durch neue Knoten mitgeteilt werden, so dass die P2P-Schicht darauf reagieren kann. Durch das Callback-Interface werden auch eingehende Nachrichten und Arbeitsaufträge an die P2P-Ebene nach oben gereicht.

Die Proxy-Ebene ist auch dafür zuständig, die Job-Verteilung zu unterstützen. So muss ein HTTP-Server, der den Anwendungscode sowie Parameterdateien vorhält, hier gestartet werden. Dieser Server wird dann auch der Empfänger für Ergebnisse der Teilberechnungen sein.

3.4.3. P2P-Ebene

Die P2P-Ebene ist die wichtigste und umfangreichste Ebene in der Middleware und setzt auf der Proxy-Schicht auf. Sie sorgt für das Erstellung und Pflege einer Topologie in Form von Nachbarschaftslisten. Die P2P-Schicht stellt u.a. Funktionen bereit, um mit anderen Knoten des P2P-Netzes zu kommunizieren, dem Netz beizutreten, das Netz zu verlassen, Jobs zu starten, Aufträge zu verteilen oder neue Knoten im Netz zu akzeptieren oder abzulehnen.

Einige Ihrer Funktionen werden konzeptionell nur von Clients, d.h. von beliebig gearteten Benutzerschnittstellen, andere von den Anwendungen, die verteilt ausgeführt werden, genutzt.

3.4.4. Anwendungsschicht

Die Anwendungsschicht dient zur Verwaltung der verteilten Anwendung. Über diese Schnittstelle kommuniziert die verteilte Anwendung mit der Middleware und bekommt dort alle Funktionalität zur Verfügung gestellt, die sie benötigt.

So kann die Anwendung generische Nachrichten an einen einzelnen anderen Knoten (Unicast), an alle (Broadcast) oder an eine zufällige Menge gegebener Größe (Multicast) verschicken. Die Nachrichten-Schnittstelle umfasst das Senden von simplen Byte-Arrays sowie von ganzen Java-Objekten.

Um eine Nachricht gezielt an einzelne Knoten verschicken zu können, braucht die Anwendung die Adresse des Empfängers; diese kann sie ermitteln, indem sie die Anwendungsschicht nach der Liste aller Nachbarn befragt. Diese Liste enthält die Knoten, die am gleichen Job arbeiten. Das Abschicken eines Broadcasts hat semantisch den gleichen

Effekt, als ob einzelne Nachrichten an alle Einträge dieser Nachbarschaftsliste geschickt würden. Die Broadcast-Schnittstelle macht an dieser Stelle keine Annahmen darüber, wie effizient dieser Broadcast ausgeführt wird. Im einfachsten Fall wird eine einzelne Nachricht an jeden einzelnen anderen Knoten verschickt.

Auf dieser Schicht werden Knoten nicht mehr über IP-Adressen identifiziert, sondern nur über eindeutige IDs im P2P-Netz. Diese eindeutigen IDs sollen zufällig und gleichverteilt in einem ausreichend großen Adressraum liegen.

3.4.5. Clientschicht

Die Clientschicht dient der Vereinfachung der Schnittstelle für den lokalen Nutzer der Middleware. Zur Vereinfachung für den Nutzer können für verschiedene verteilte Anwendungen jeweils eigene Clients bereitgestellt werden, welche über diese Schnittstelle mit den tiefer liegenden Schichten der Middleware kommunizieren können. Zu den dort möglichen administrativen Aufgaben gehören primär das Instanziieren und Beenden der Middleware, sowie das Verbinden und Beenden der Verbindung zum P2P-Netz.

Während des Betriebs der Middleware sind Funktionen an dieser Schnittstelle primär das Starten und Abbrechen von verteilten Anwendungen sowie der Empfang von Rückmeldungen über den Status der aktuell ausgeführten Anwendung.

3.4.6. Proxy-Manager

Der Proxy-Manager dient dazu, eingehende Proxy-Wünsche von anderen Clients zu bearbeiten. Jede Middleware-Instanz kann als Proxy-Server für andere Instanzen dienen, die sich dazu vorher anmelden müssen. Nachrichten an Proxy-Clients werden über die normale Schnittstelle zum P2P-Netz angenommen und müssen dann – statt lokal verarbeitet zu werden – an den Proxy-Client weitergeleitet werden. Dadurch ist es Knoten, die aus Sicht des P2P-Netzes hinter einer Firewall stehen, möglich, an den verteilten Berechnungen teilzunehmen. Mit dem Proxy-Manager wäre es mit Erweiterungen auch möglich, eine Superpeer-Topologie zu realisieren [YGM03].

Beim Superpeer-Konzept werden Knoten in zwei Sorten geteilt, in Superpeers und „normale“ Knoten. Die normalen Knoten kommunizieren im Normalfall jeweils nur mit Superpeers bzw. setzen bezogen auf verteilte Datenspeicherung Suchanfragen nur über die Superpeers ab. Suchanfragen werden nur auf den Superpeers verarbeitet, diese halten die Dateilisten Ihrer Clients vor. So wird vermieden, dass jeder Knoten mit Suchanfragen „bombardiert“ wird. Zur Konstruktion von Superpeer-Topologien siehe auch [Mon04].

3.5. Konzept der geplanten Middleware

In diesem Abschnitt soll dargelegt werden, wie gewisse Aktionen innerhalb der spezifizierten Strukturen ablaufen sollen.

3.5.1. Starten der Middleware

Das Starten der Middleware kann auf drei verschiedene Arten erfolgen.

Start von „außen“: Eine dedizierte Startkomponente startet nach und nach die einzelnen Instanzen der verschiedenen Protokollebenen und verknüpft diese dann miteinander.

Start von „unten“: Der Nutzer initiiert die Proxy-Ebene und diese startet im Rahmen ihres Konstruktors die nächsthöhere Protokolleinheiten. Dies wird fortgesetzt bis hin zur Client-Schicht.

Start von „oben“: Der Nutzer startet seine Benutzerschnittstelle, welche daraufhin die nächsttiefere Ebene initiiert. Dies wird bis zur Proxy-Schicht kaskadiert.

In diesem Prototypen soll die dritte Variante Verwendung finden. Es scheint aus Sicht des Nutzers die natürlichste Variante, wenn dieser seinen Client startet – also nur die Instanz, mit der er sich auch beschäftigen will – und der Rest geschieht intern.

3.5.2. Verbinden mit dem P2P-Netz

Die Aufbau einer Verbindung zu einem bestehenden P2P-Netz soll durch Senden einer Nachricht an einen Teilnehmer des Netzes erfolgen. Der Zielknoten muss dafür keine ausgezeichnete Rolle einnehmen, er muss nur selber aktiv im P2P-Netz sein. Von diesem Knoten soll der Neuankömmling dann seine initiale Nachbarschaftsliste erhalten.

Die Frage, wie ein neuer Knoten die Adresse eines aktiven Teilnehmers bekommt, soll hier offen bleiben. Es wird bewusst kein zentraler Einstiegsknoten geschaffen, da dieser ein System anfällig für die üblichen Zentralisierungsprobleme (Skalierbarkeit, Robustheit) macht.

Denkbar wäre eine Mitnutzung des DNS-Systems (Domain Name Service), um den Namen eines zentralen Zugangspunktes zu verschiedenen aktuell gültigen IP-Adressen aufzulösen.

3.5.3. Unbelasteter Fall

Im unbelasteten Fall – d.h. wenn kein rechenbereiter Auftrag vorliegt – sorgt die P2P-Schicht dafür, dass sie auf einen eingehenden Auftrag vorbereitet ist und pflegt, je nachdem ob das für den konkreten Algorithmus des P2P-Netzes relevant ist, ihre Nachbarschaftslisten. Im Fall eines epidemischen Protokolls müssen in regelmäßigen Zeitabständen Nachbarschaftslisten mit zufällig gewählten bekannten Knoten ausgetauscht werden, so dass sich eine Änderung an der virtuellen globalen Nachbarschaftsliste durch das Netz propagiert.

3.5.4. Starten von Aufträgen

Das manuelle Starten von Aufträgen durch den Nutzer erfolgt in der Client-Schicht. Die Spezifikation des Auftrags wird weiter an die P2P-Schicht gereicht. Dort muss als erstes dafür gesorgt werden, dass der Auftrag verteilt werden kann, d.h. dass Code und Parameter heruntergeladen werden können. Dieser Dienst wird von der Proxy-Schicht bereitgestellt.

Im Anschluss kann die P2P-Schicht durch Übergabe der Auftragsspezifikation an eine neue Instanz der Anwendungsschicht die lokale Verarbeitung des Auftrags anstoßen. Dann kann die Verteilung des Auftrags beginnen. Die genaue Vorgehensweise ist von den realisierten Algorithmen in der P2P-Schicht abhängig. Für ein epidemisches Protokoll wird in regelmäßigen Abständen die Hälfte des aktuellen Arbeitspaketes an einen zufälligen Nachbarn verschickt. Dies erfolgt so lange, bis alle Ergebnisse zurück beim Initiator angekommen sind.

3.5.5. Empfangen von Aufträgen

Wenn ein neuer Auftrag über das P2P-Netz empfangen wird, muss als erstes geprüft werden, ob der Auftrag überhaupt bearbeitet werden soll. Wenn dies nicht der Fall ist, kann er an dieser Stelle schon verworfen werden soll. Die Kriterien für die Annahme eines Auftrags können vielfältig sein. Eine einfache Variante ist es, jeden Auftrag anzunehmen, sofern der lokale Knoten noch nicht mit einer Berechnung beschäftigt ist.

Eine eingegangene und akzeptierte Auftragsspezifikation wird primär wie ein lokal gestarteter Auftrag behandelt. Zur Lastverteilung wird der Anwendungscode heruntergeladen und auf einem lokalen Server zur Verteilung bereitgehalten, der entsprechende Eintrag für die URL des Anwendungscodes in der Auftragsspezifikation wird modifiziert.

Sobald der Auftrag auch vom lokalen Knoten heruntergeladen werden kann entspricht das weitere Vorgehen dem oben genannten. Eine Instanz des Auftrags wird lokal gestartet, der Rest wird durch den Verteilungsalgorithmus der P2P-Schicht in das P2P-Netz propagiert.

3.5.6. Abliefern von Ergebnissen

Eine verteilt ausgeführte Berechnung macht nur dann Sinn, wenn die Ergebnisse der Berechnung auch wieder ihren Weg zurück zum Initiator finden. Dafür muss in der Anwendungsschicht ein Dienst bereitgestellt werden, den die verteilte Anwendung nutzen kann.

Das Abliefern der Ergebnisse erfolgt ähnlich wie das Herunterladen von Anwendungscode und Parametern. Der HTTP-Server, der dafür beim Initiator zur Verfügung steht (siehe Abschnitt 3.4.2), dient dabei auch als Zielort für die Ergebnisse. Die Middleware-Instanzen der Rechenknoten müssen dafür sorgen, dass die von der Anwendung erhaltenen Ergebnisdaten dorthin übertragen werden.

3.5.7. Versand von Anwendungsnachrichten

Das Versenden von Nachrichten durch die Anwendung erfordert nur wenige Schritte. Nachrichten an einzelnen Knoten (in Form von Byte-Arrays oder einem Objektbaum) werden auf der P2P-Schicht und der Proxy-Schicht jeweils in entsprechende Nachrichtentypen verpackt, serialisiert und dann als UDP-Paket versandt.

Wie der auf der Anwendungsschicht angebotene Broadcast und Multicast realisiert wird, kommt wieder auf die konkrete Implementierung der jeweiligen Schichten an. Der simple Ansatz schickt eine einzelne (Unicast-) Nachricht an jeden der Empfänger. Bei vielen bekannten Nachbarknoten skaliert dieser Ansatz jedoch schlecht. Ein ausgefeilterer Ansatz verwendet effiziente Broadcast- bzw. Multicast-Algorithmen.

3.5.8. Empfang von Anwendungsnachrichten

Der Empfang von Anwendungsnachrichten durch die verteilte Anwendung erfolgt, indem eine eingehende Nachricht in den jeweiligen Nachrichtentypen der Proxy-Schicht und der P2P-Schicht gekapselt empfangen wird und die Anwendungsnachricht schließlich von der P2P-Schicht an die Anwendungsschicht übergeben wird. Diese kann die Nachricht dann passend dekodieren (je nachdem ob ein Byte-Array oder ein Objektbaum versendet wurde) und der verteilten Anwendung übergeben.

4. Implementierung

In diesem Kapitel wird eine konkrete Implementierung für das im vorherigen Kapitel vorgestellte Framework präsentiert. Es wird insbesondere auf die Besonderheiten und essentiellen Punkte der im Rahmen dieser Arbeit erstellten Implementierung eingegangen.

4.1. Allgemeines

Im Folgenden werden kurz ein paar allgemeine Techniken vorgestellt, die im Rahmen der Implementierung der Middleware häufiger oder vergleichbar in verschiedenen Bereichen eingesetzt wurden.

4.1.1. Nachrichten

Jede Ebene, auf der Nachrichten verschickt werden, kann ihre eigenen Nachrichtentypen implementieren. So gibt es auf der Proxy-Ebene die *P2PMessage*, auf der P2P-Ebene für das epidemische Protokoll (siehe Abschnitt 4.7) die *EpidemicMessage* und auf der Anwendungsebene die *ApplicationMessage*. Die implementierten Nachrichtenklassen werden später noch genauer beschrieben. Nachrichtenobjekte der Proxy-Ebene, d.h. Instanzen der Nachrichtenklasse P2PMessage, werden immer in serialisierter Form, also als Array von Bytes über das Netzwerk übertragen und beim Empfänger wieder deserialisiert.

4.1.2. Nachrichtenwarteschlangen

Jede Schicht, in der Nachrichten verarbeitet werden (d.h. Proxy-Schicht, P2P-Schicht, Anwendungsschicht), unterhält eine eigene Nachrichtenwarteschlange für eingehende Nachrichten, welche aus dem Basisnetzwerk empfangen wurden. Sobald die Verarbeitung einer Nachricht in einer Schicht abgeschlossen ist und feststeht, dass der Inhalt der Nachricht an die nächste Schicht weitergegeben werden muss, wird dieser Inhalt (ein serialisiertes Objekt der Nachrichtenklasse der empfangenden Schicht) in die Nachrichtenwarteschlange dieser Schicht eingestellt. Dort wird diese dann später ausgelesen und weiterverarbeitet.

4.1.3. Threads, Synchronisation und Nebenläufigkeit

Damit empfangene Nachrichten möglichst schnell beantwortet bzw. bearbeitet werden können, soll ein größtmögliches Maß an Nebenläufigkeit erreicht werden. Allerdings gibt es zahlreichen Passagen in der Bearbeitung von Nachrichten, die nur ein Thread gleichzeitig bearbeiten darf, um keine konkurrierenden Zugriffe auf die gleichen Daten zu verursachen. Primär existiert in jeder Ebene ein Thread, der für die Bearbeitung von eingehenden Nachrichten in dieser Schicht zuständig ist. Dieser Thread wartet so lange, bis eine Nachricht in die Nachrichtenwarteschlange dieser Schicht eingereiht wird, nimmt diese dann entgegen, bearbeitet sie und wartet dann auf die nächste Nachricht. Diese „Arbeits-Threads“ für die Abarbeitung von empfangenen Nachrichten auf jeder der Nachrichtenschichten können sich untereinander nicht beeinflussen und können deswegen vollständig nebenläufig ablaufen. Einziger Kollisionspunkt sind die Lese- und Schreiboperationen auf den Warteschlangen, diese werden aber synchronisiert.

Allerdings gibt es noch einige andere Threads, die in der Middleware existieren und teilweise synchronisiert werden müssen. So werden alle Aktionen, welche vom Benutzer ausgelöst werden (bspw. durch eine grafische Benutzerschnittstelle), in einem oder mehreren eigenen Threads bearbeitet. Diese können aber mit den Empfangs-Threads in Konflikt geraten. Ebenso wird eine in der Middleware ablaufende verteilte Anwendung in einem eigenen Thread verarbeitet. Von ihr ausgelöste Sendevorgänge überschneiden sich mit den Empfangs-Threads. Desweiteren können in den verschiedenen Schichten weitere aktive Komponenten vorhanden sein, die eigenständig Nachrichten senden (z.B. der *CyclicSender* des epidemischen Protokolls in Abschnitt 4.7.5).

Man erkennt an dieser Stelle also, dass der vorhandene Code nach kritischen Abschnitten analysiert werden muss und diese durch Synchronisation vor konkurrierendem Zugriff geschützt werden müssen. Auf diese Fälle wird in den entsprechenden Abschnitten eingegangen.

4.1.4. Konfigurationsdatei

Um möglichst flexibel auf verschiedene Einsatzzwecke reagieren zu können, müssen die Basiskomponenten der Middleware konfigurierbar sein. Dazu bietet Java Unterstützung durch sog. Properties (*java.util.Property*), welche auch persistent, z.B. in einer reinen ASCII-Datei in einem lesbaren Format gespeichert werden können. Alle Optionen können dann aus dieser Datei eingelesen werden und von einem Property-Objekt abgefragt werden. Die verschiedenen möglichen Konfigurationsoptionen finden sich vollständig im Anhang A.

4.1.5. Serialisierung

Java unterstützt einen relativ einfachen Mechanismus zum Vorbereiten von Objekten für die Speicherung oder das Versenden über ein Netzwerk, der die Objekte intern in eine Byte-Folge konvertiert. Dazu bietet Java das Interface *Serializable* an, dessen Implementierung die Möglichkeit signalisiert, dieses Objekt in einen Bytefolge konvertieren zu können. Die Serialisierung erfolgt üblicherweise vollständig durch die Java-Virtual-Machine, beim Schreiben der Objekte werden dabei Informationen über die Objektklasse und den Inhalt der Datenfelder geschrieben.

Java bietet jedoch auch die Möglichkeit an, eigene Serialisierungsmethoden zu definieren, um etwa den Inhalt von Variablen vor der Serialisierung noch zu modifizieren. Dazu können die Methoden *readObject()* und *writeObject()* verwendet werden, in denen der Programmierer dann beliebige Aktionen tätigen kann. Auch hierbei wird vom System automatisch ein Header für das Objekt erzeugt, der zumindest Informationen über die verwendete Klasse und die Version der Implementierung enthält. Das Serialisieren einen Objektes erfolgt üblicherweise über die Methode *writeObject()* des *ObjectOutputStreams*, das zugehörige Deserialisieren, d.h. das Instanziieren eines Objektes aus seiner persistenten Form, erfolgt mittels *readObject()* auf dem *ObjectInputStream*.

Die meisten Objekte, die im Rahmen des Sendens von Nachrichten in der Middleware serialisiert werden, haben ganz eigene Methoden zur Serialisierung, ohne den Java-Serialisierungs-Mechanismus zu verwenden. Dies hat den Grund, dass ein großer Wert darauf gelegt wurde, dass die (regelmäßig zu verschickenden) Nachrichten möglichst klein sind. Dies lässt sich optimieren, indem der Serialisierungsheader, der von Java verwendet wird, weggelassen wird und auch die Datenfelder ohne Feldtrennzeichen im Bytestrom kodiert werden.

4.1.6. Nachrichtengröße

In der vorliegenden Implementierung ist die Größe der verschickbaren Nachrichten durch die Middleware selber nicht beschränkt. Jedoch wird als Basisprotokoll UDP eingesetzt, wobei im vorliegenden Modell keine Segmentierung implementiert wurde. Dies bedeutet, dass zu versendende Nachrichten mit einer Größe von mehr als 64 Kilobyte aufgrund der Beschränkungen von UDP im Moment nur unvollständig versendet würden. Deshalb wird das Versenden solch großer Nachrichten im Moment nicht unterstützt und solche Pakete werden verworfen. Die Anwendung wird davon derzeit nicht benachrichtigt.

4.1.7. Logging

Um die Erstellung und Wartung der Implementierung zu gewährleisten und eine Möglichkeit zu bieten, die internen Abläufe der realisierten Algorithmen überprüfen zu können, wurde mittels der Java-eigenen Logging-Mechanismen (*java.util.logging.**) der komplette Code mit entsprechenden Anweisungen für das Logging auf verschiedenen Stufen versehen.

4.2. P2PProxy

Die Klasse *P2PProxy* ist der Hauptteil der Implementierung der Proxy-Ebene (siehe 3.4.2). Sie implementiert das Interface *ProxyLayer* und stellt allgemeine Basisdienste für verschiedene P2P-Protokolle zur Verfügung. Dazu gehören der Nachrichten-Versand, das Betreten und Verlassen eines P2P-Netzes sowie das Starten eines Webservers zum Bereitstellen von Anwendungscode und Parameterdateien sowie zum Empfang von Lösungen (siehe *DistributionManager* in Abschnitt 4.5). Außerdem wird hier entschieden, ob Nachrichten über einen Proxy-Server verschickt bzw. empfangen werden oder auf direktem Wege; ebenso wird ein eventuell lokal bereitgestellter Proxy-Server (siehe auch Abschnitt 4.3) gestartet.

4.2.1. Adressierung

Die Adressierung von Knoten erfolgt innerhalb der Proxy-Schicht mittels der *NetworkId*. Die NetworkId identifiziert Knoten eindeutig innerhalb des Internets. Aus ihr lassen sich direkt IP-Adresse und UDP-Port des Knotens ablesen. Desweiteren enthält die NetworkId eine laufende Nummer, welche als Unterscheidungskriterium für Proxy-Clients dient. Die laufende Nummer 0 wird immer für den lokalen Knoten verwendet, weitere laufende Nummern werden in aufsteigender Reihenfolge an Proxy-Clients vergeben.

4.2.2. Initialisierung

Laden der Konfiguration

Bei der Initialisierung wird als erstes die Konfigurationsdatei (siehe 4.1.4) geladen. Die persistente Form der Konfiguration wird ausschließlich vom *P2PProxy* verwaltet, alle anderen Klassen, die Konfigurationsinformationen benötigen, bekommen diese vom *P2PProxy*. Dazu stellt dieser die Methode *getConfig()* bereit.

Proxy-Verbindung

Im Anschluss daran wird, sofern konfiguriert, eine Verbindung zum Proxy-Server aufgebaut und von diesem die eigene NetworkId empfangen. Falls laut Konfiguration gefordert, wird dann der lokale Proxy-Server gestartet.

Binden des UDP-Sockets und ermitteln der NetworkId

Nun wird der laut Konfiguration zu nutzende UDP-Socket gebunden und (sofern noch nicht durch Aufbau einer Proxy-Verbindung zu einem Proxy-Server geschehen) eine NetworkId berechnet.

Starten von Threads

Abschließend werden noch (für den nicht-Proxy-Fall) der Thread gestartet, der Pakete vom UDP-Socket empfängt und in die Warteschlange einreiht, sowie derjenige, der die Nachrichten aus der Warteschlange nimmt und verarbeitet. Dieser Vorgang ist in zwei Teile getrennt, damit die Zeit, in der gerade kein Empfänger tatsächlich am Socket empfängt, möglichst kurz ist. Im Proxy-Fall wurde bereits beim Aufbau der Verbindung zum Proxy-Server ein eigener Thread gestartet, der blockierend auf dem TCP-Kanal zum Proxy-Server Pakete empfängt und diese dann verarbeitet. In diesem Fall wird also der Umweg über die Warteschlange übergangen, da auf dem TCP-Kanal keine Nachrichten verloren gehen können.

4.2.3. Senden von Nachrichten

Das Senden von Nachrichten erfolgt prinzipiell so, dass dem P2PProxy eine serialisierte Nachricht (als *byte[]*) und ein Empfänger in Form einer *NetworkId* übergeben werden. Daraufhin wird die Nachricht in eine *P2PMessage* verpackt, Empfänger und Absender hinzugefügt, die Nachricht serialisiert, in ein UDP-Datagram verpackt und dieses dann an den Empfänger gesendet. Für den Standardfall des unbestätigten Sendens reichen diese Schritte auch schon aus.

Da Sendeoperationen prinzipiell von mehreren Threads gleichzeitig ausgeführt werden können und sich nicht beeinflussen dürfen, jedoch Empfangsoperationen parallel dazu durchgeführt werden können und sollen, werden alle Sendeoperationen gegen ein gemeinsames „Sendeobjekt“ synchronisiert und nicht gegen den *P2PProxy* selber. So werden nur die Sendeoperationen beeinflusst. Empfangsoperationen werden gegen das P2PProxy-Objekt selber synchronisiert, obwohl nebenläufige Verarbeitung von Paketen im Normalfall hier nicht vorkommen kann.

Einen beispielhaften unbestätigten Sendevorgang kann man in Abbildung 4.1 erkennen.

Die angegebenen Message Sequence Charts (MSC) sollen den schematischen Ablauf darstellen; die angegebenen Parameterlisten müssen deshalb nicht ganz der Realität entsprechen. Insofern können einzelne Parameter, die für das Verständnis nicht relevant sind, fehlen.

Bestätigtes Senden

Beim bestätigten Senden kommen noch einige Schritte hinzu. Beim Vorbereiten einer Nachricht bekommt diese eine Sequenznummer. Nach dem Senden der Nachricht wird bis zu einer in der Konfigurationsdatei festgelegten Anzahl von Millisekunden gewartet (d.h. der Thread schläft), ob eine Bestätigung für diese Nachricht eingetroffen ist.

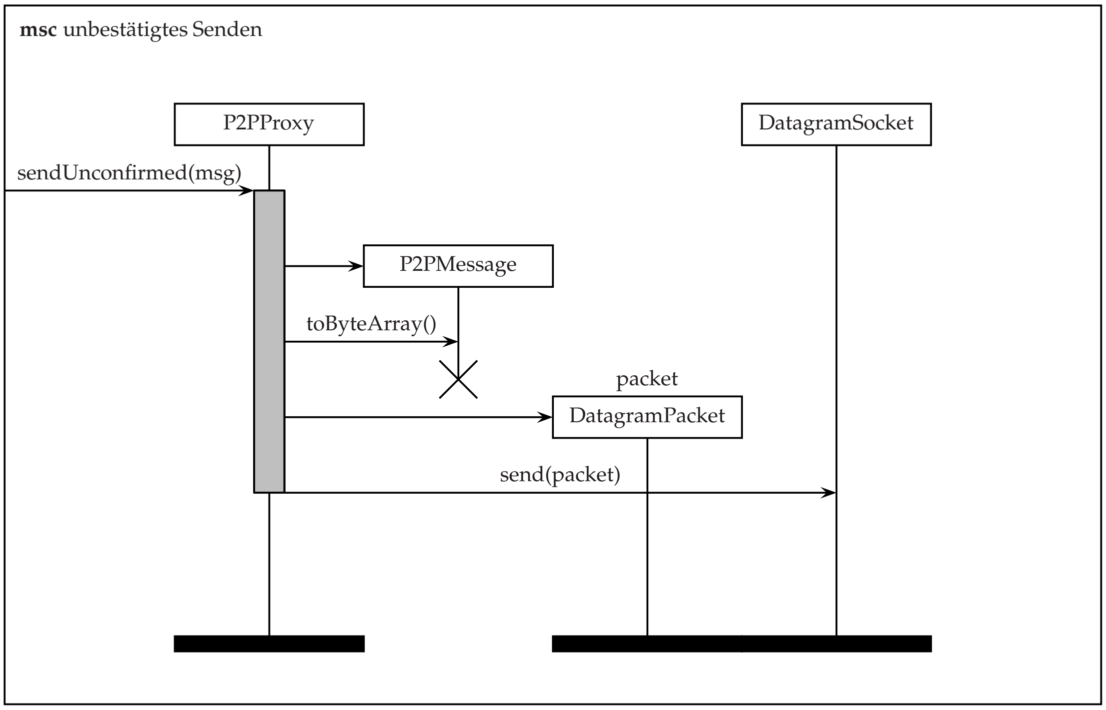

Abbildung 4.1.: Unbestätigtes Senden von Nachrichten

Dazu wartet der Sende-Thread auf eine Benachrichtigung vom Empfangsprozess. Sollte die Quittung ausbleiben, wird das Senden der Nachricht bis zu einer konfigurierbaren Anzahl von Versuchen wiederholt. Sollte auch bis zum letzten Versuch keine Quittung eingegangen sein, so wird die Nachricht verworfen. Die sendende Schicht erhält hierüber jedoch keine Information. Deswegen kann man streng genommen nicht von einem verlässlichen Nachrichtendienst sprechen.

Die Folge der Methodenaufrufe beim bestätigten Senden ist in Abbildung 4.2 zu erkennen.

Senden über den Proxy

Wenn Pakete nicht direkt ins Netz, sondern über den Proxy-Server verschickt werden, so wird einfach das serialisierte Datenpaket in den TCP-Stream zum Proxy-Server gelegt, dort gelesen und dann über den dortigen UDP-Port verschickt. Das Paket muss dort nur in ein DatagramPacket verpackt werden und kann sofort verschickt werden.

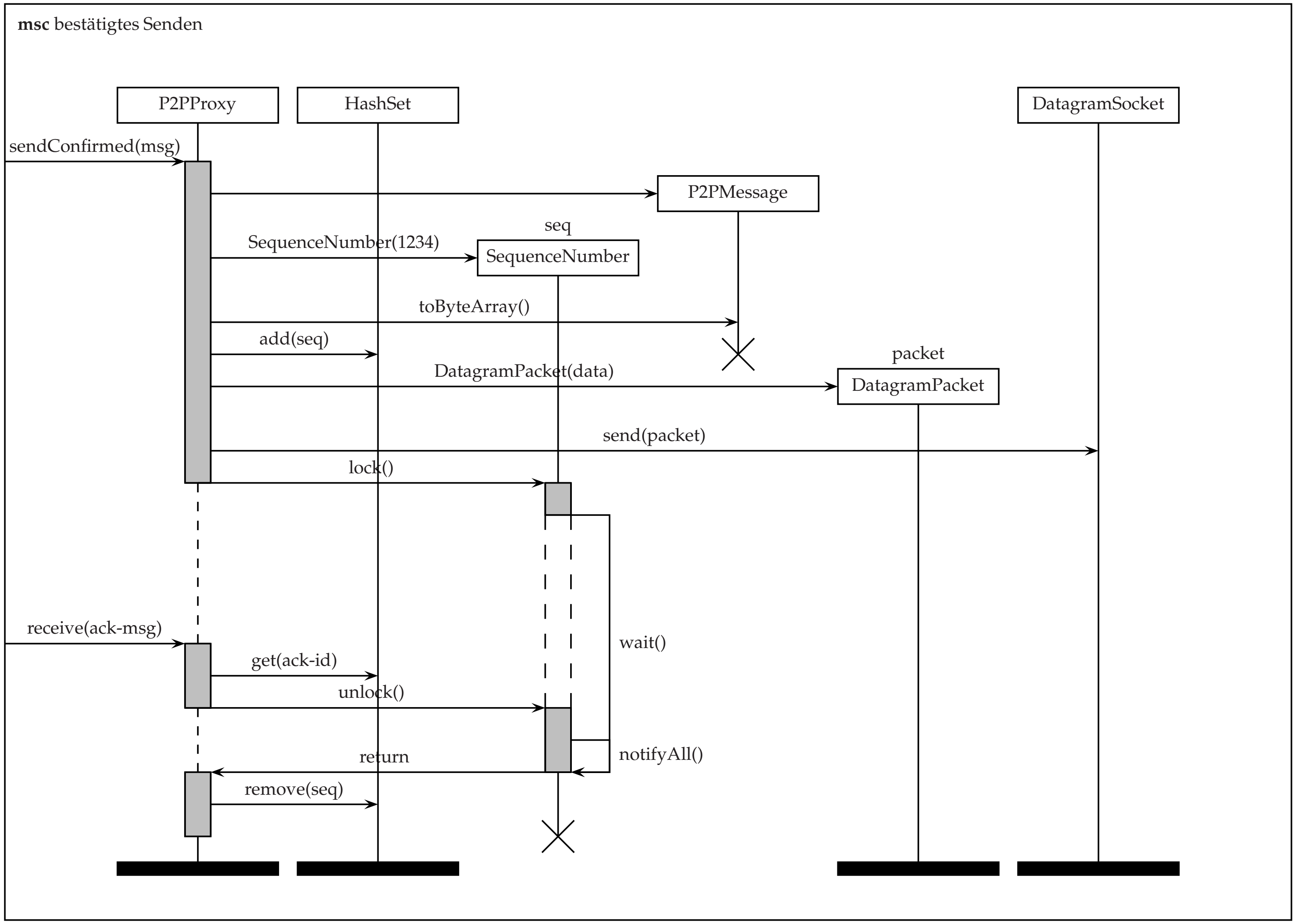

Abbildung 4.2.: Bestätigtes Senden von Nachrichten

4.2.4. Empfangen von Nachrichten

Lesen vom Socket

Wie bereits weiter oben erwähnt existiert ein eigener Thread, der in einer Schleife auf der *receive()*-Methode des UDP-Socket blockiert, bis eine Nachricht eingeht. Aus den eingehenden Binärdaten wird eine *P2PMessage* deserialisiert, welche dann in die Eingangswarteschlange gestellt wird.

Das Zusammenspiel zwischen den involvierten Klassen beim Nachrichtenempfang ist in Abbildung 4.3 zu sehen. Der Empfang von Quittungen ist bereits in Abbildung 4.2 enthalten.

Verarbeiten von Nachrichten

Ein weiterer Thread blockiert so lange auf der Warteschlange, bis eine neue Nachricht verfügbar wird. Diese wird daraufhin an den *P2PProxy* übergeben. Hier wird zuerst der Nachrichtentyp ausgelesen, woraufhin eine spezifische Bearbeitung erfolgen kann. Ein Teil der Nachrichtentypen wird nicht für eine Funktionalität der Proxy-Schicht benötigt, sondern stellt nur eine Vereinfachung für die P2P-Schicht dar. Diese Nachrichten werden direkt in einen Methodenaufruf in der P2P-Schicht umgewandelt (z.B. JOIN, ACCEPT). Der Vorteil dabei ist, dass nicht jedes P2P-Protokoll diese Nachrichtentypen selber implementieren muss.

Eine besondere Behandlung bekommen der Empfang einer bestätigten Nachricht und der Empfang einer Quittung: Bei Empfang einer bestätigten Nachricht muss zum Einen eine Quittung an den Absender verschickt werden, dass die Nachricht angekommen ist (die Identifizierung erfolgt durch die Sequenznummer). Außerdem wird die Nachricht nur dann an die nächste Schicht weitergeleitet, wenn diese Nachricht nicht bereits vorher empfangen wurde (z.B. weil die Quittung im Netz verloren gegangen ist und der Sender die Nachricht wiederholt hat). Somit werden Duplikate eliminiert.

Bei Empfang einer Quittung muss die entsprechende Sequenznummer aus der Liste entfernt werden, die speichert, welche Nachrichten noch ausstehen und der entsprechende Sendeprozess benachrichtigt werden, der auf die Quittung wartet. Damit ist der Sendevorgang abgeschlossen.

Empfangen von Nachrichten bei Proxy-Clients

Wie oben bereits erwähnt wird die Nachrichtenwarteschlange für eingehende Nachrichten beim Empfang als Proxy-Client umgangen. In diesem Fall verarbeitet der empfangende Thread direkt die über den TCP-Kanal vom Proxy-Server eingehende Nachricht durch den Aufruf der receive-Methode am P2PProxy. Ein Beispiel-Verlauf ist in Abbildung 4.4 veranschaulicht.

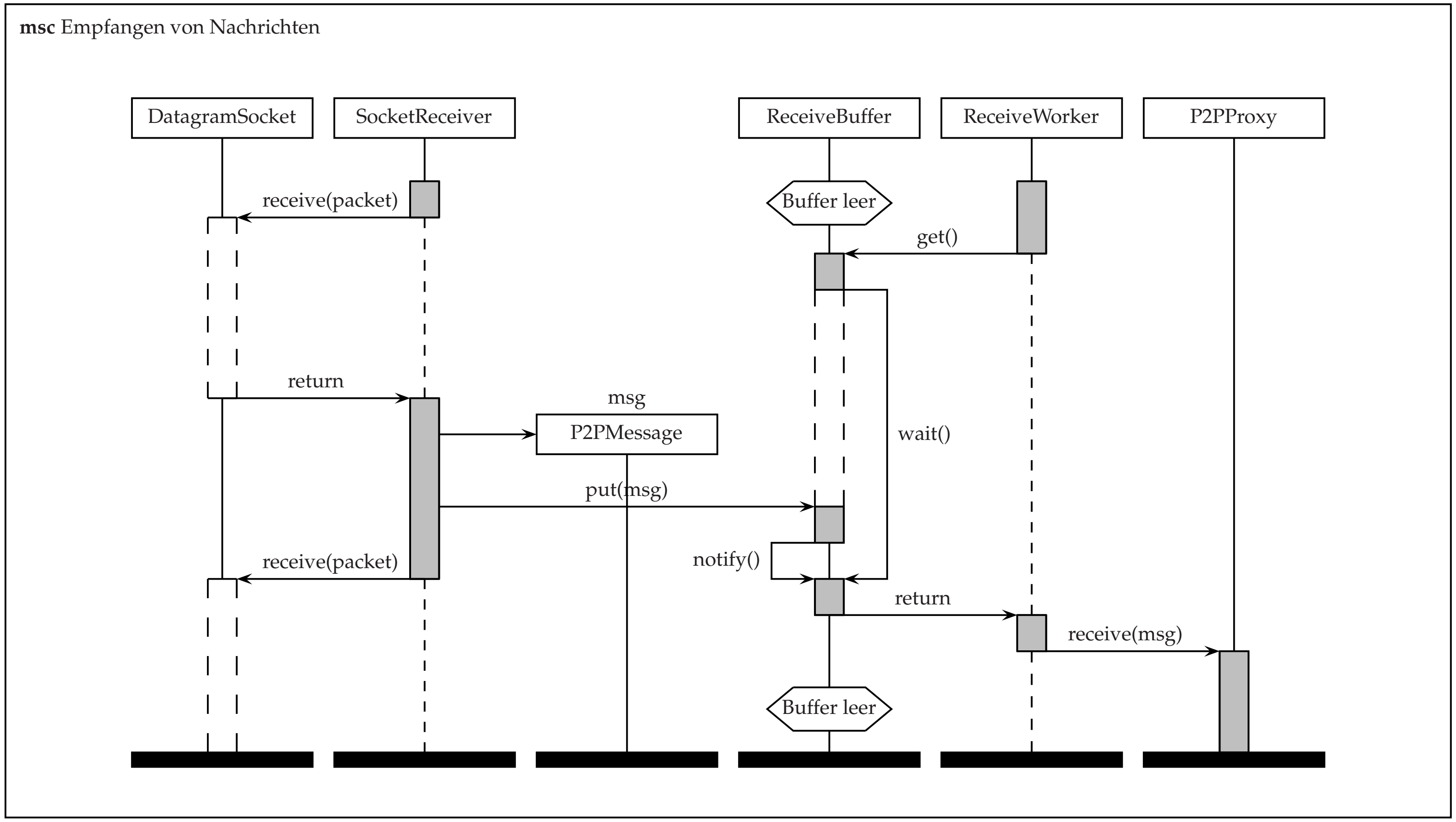

Abbildung 4.3.: Empfang einer Nachricht

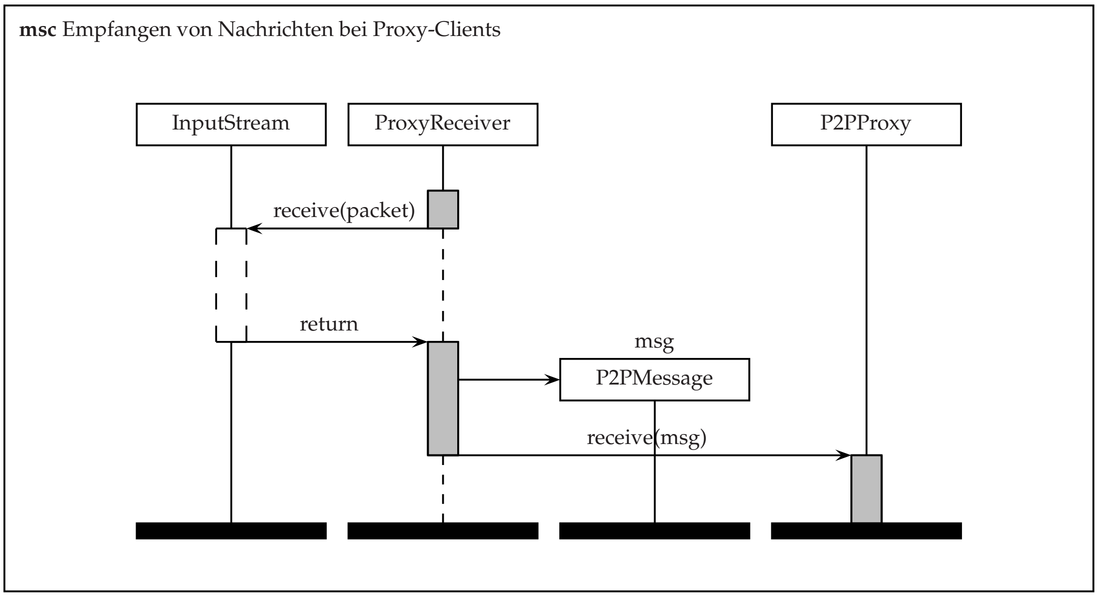

Abbildung 4.4.: Empfang einer Nachricht als Proxy-Client

4.2.5. Verteilung von Aufträgen starten

Auf der Proxy-Ebene wird auch der Webserver realisiert, der die Download-Möglichkeit für Anwendungscode und Parameterdateien zur Verfügung gestellt. Da der *P2P-Proxy* die Schnittstelle zu dieser Schicht realisiert, muss diese Funktionalität entweder dort realisiert oder an eine andere Instanz delegiert werden. Dazu gibt es den *DistributionManager* (siehe auch 4.5). Da zu einer Zeit nur ein Job gleichzeitig lokal verarbeitet werden soll, wird auch nur ein Job gleichzeitig verteilt. Somit wird beim Starten eines neuen Managers die alte Instanz beendet und ein neuer *DistributionManager* initialisiert. Dort wird auch das Beschreibungsobjekt für die aktuelle Berechnung generiert und dann an das P2P-Protokoll weitergegeben.

Der Start eines DistributionManagers wird in Abbildung 4.5 gezeigt.

4.3. SimpleProxyManagerImpl

Die Klasse *SimpleProxyManagerImpl* ist, wie der Name schon suggeriert, eine recht einfache Implementierung eines ProxyManagers, wie er als Interface definiert wurde. Der ProxyManager ist dazu da, die verschiedenen Threads und Kommunikationsendpunkte zu allen Proxy-Clients zu verwalten und neu eingehende Verbindungswünsche von weiteren Rechnern, die diesen Knoten als Proxy-Server verwenden wollen, zu bearbeiten.

Der *SimpleProxyManagerImpl* ist als eigener Thread implementiert, da der Manager unabhängig von allen anderen Middleware-Aktionen agieren muss.

4.3.1. Initialisierung

Bei der Initialisierung des ProxyManagers werden primär die benötigten Konfigurationsparameter (TCP-Port des Proxy-Servers und UDP-Port der Middleware) ausgelesen und eine Tabelle zum Abbilden von Zieladressen (NetworkIds) auf Threads initialisiert.

4.3.2. Starten des Threads

Beim Starten bindet der Manager erstmals den TCP-Port, der laut Konfiguration benutzt werden soll. Wenn in der Konfiguration kein Port definiert ist, wird ein beliebiger freier Port benutzt. In einer Schleife wird dann blockierend auf neue Verbindungsversuche von potentiellen Clients gewartet. Alle fünf Sekunden wird jedoch die Blockierung beendet, um zu prüfen, ob der Proxy-Server beendet werden soll (z.B. weil die Jobverteilung abgeschlossen ist oder der ganze Auftrag vom Initiator abgebrochen wurde).

4.3.3. Aufbau von Proxy-Sitzungen

Dieser einfache Proxy-Manager lehnt eingehende Verbindungswünsche niemals ab, sondern akzeptiert jeden neuen Client. Wenn dieser sich verbindet, wird vom Proxy-Manager ein neuer Thread (*SimpleProxyThread*) gestartet, der für die Abwicklung der Kommunikation mit diesem Client verantwortlich ist. Daraufhin berechnet der Proxy-Manager eine neue NetworkId für den Client auf Basis der IP-Adresse des Proxy-Servers und schickt diese dem Client über den aufgebauten TCP-Kanal.

Die Hauptaufgabe des ProxyThread ist es ab hier, blockierend auf dem InputStream der TCP-Verbindung zu lesen und eingehende Pakete vom Client entgegenzunehmen und über den Proxy-Manager weiterzuleiten. Dies bietet die Möglichkeit, dass ein Client seine Nachrichten nicht nur über den Proxy-Server empfängt, sondern auch Nachrichten über den Proxy-Server verschicken kann. Dies ist nötig für Clients, die sich hinter sehr restriktiven Firewalls befinden, die auch ausgehenden Netzwerkverkehr filtern.

4.4. Beschreibungsdatei für Aufträge

Das sog. *WorkSpecFile* dient dazu, Aufträge zu beschreiben, um sie im P2P-Netz verteilen zu können. In so einem Objekt sind die URLs aller für die Ausführung einer verteilten Anwendung benötigten Dateien, d.h. diejenige des Anwendungscodes, von eventuellen Parameterdateien und von dem Ort, an den die Ergebnisse nach Beendigung der Berechnung geschickt werden sollen, enthalten. Außerdem enthält es den

Namen und die Id eines solchen Auftrags, sowie die Anzahl der Teilaufträge, in die der Auftrag geteilt werden soll und den Typ eines Auftrags.

Ein Auftrag kann die bereits in Abschnitt 3.1 angesprochenen Arten einer Berechnung haben. In der konkreten Implementierung werden drei Arten von Berechnungen unterschieden: Aufträge mit einem Parameter pro Teilauftrag, Aufträge mit n Teilaufträgen aber den gleichen Parametern und Aufträge mit n Teilaufträgen, jedoch ganz ohne Parameter.

Das Teilen von Aufträgen auf mehrere Rechner erfolgt durch Teilen dieser Beschreibungsdatei. Dazu kann die Beschreibungsdatei durch die Methode *split()* in mehrere Teilaufträge geteilt werden, dabei werden die vorhandenen Parameterdateien in mehrere Teilmengen aufgeteilt bzw. die Anzahl der gewünschten Teilaufträge entsprechend aufgeteilt. Die neu entstandenen Teilauftragsbeschreibungen können dann weiterverschickt werden.

4.5. DistributionManager

Der *DistributionManager* ist die Implementierung eines HTTP-Servers speziell für die Zwecke der Middleware. Hier können der Anwendungscode und – falls existent – Parameterdateien zu verteilten Anwendungen mittels der GET-Methode des HTTP-Protokolls [FGM$^+$99] heruntergeladen und Ergebnisse mittels der PUT-Methode wieder heraufgeladen und damit zurück zum Initiator der Berechnung befördert werden. Konzeptionell gehört der DistributionManager in die Proxy-Ebene, damit nicht jedes P2P-Protokoll seinen eigenen Dienst zum Verteilen von Code implementieren muss.

Jeder DistributionManager ist für genau eine verteilte Berechnung (man könnte auch sagen für genau eine *Jobid*) zuständig. Auf dem Port, auf dem er Anfragen entgegen nimmt, werden nur URLs, die diese Jobid im Pfad haben, als gültig akzeptiert. Beim Starten eines Auftrags wird im DistributionManager die Beschreibungsdatei für den Auftrag erstellt, die dann von der P2P-Schicht weiterverteilt wird.

Der DistributionManager kann in zwei verschiedenen Modi laufen. Im normalen Modus stellt er nur Anwendungscode zum Download bereit und wird wieder beendet, sobald der lokale Knoten seine Berechnung an diesem Job beendet hat. Dieser Modus dient dazu, die Download-Last auf mehrere Knoten zu verteilen. Je mehr Knoten an einem Auftrag mitrechnen, desto mehr Quellen für den Anwendungscode stehen bereit. Dies dient der Skalierungsfähigkeit der Middleware.

Der DistributionManager kann auch als sog. Root-Server fungieren, er nimmt dann Ergebnisse entgegen und meldet den erfolgreichen Empfang aller ausstehenden Ergebnisse an die P2P-Schicht über den *P2PProxy* zurück. Außerdem werden auch nur in diesem Modus Parameterdateien zum Herunterladen angeboten.

Der DistributionManager des Initiators einer verteilten Anwendung ist die einzige Instanz, die einen vollständigen Überblick darüber hat, welche Teilaufträge noch ausstehen und welche Teilaufträge bereits bearbeitet und deren Ergebnisse beim Initiator angekommen sind. Versucht ein Knoten, die Parameter eines Teilauftrags herunterzuladen, dessen Ergebnisse bereits vorliegen, so wird dem Client mitgeteilt, dass diese Parameter-Datei nicht mehr existiert, so dass der Knoten diesen Teilauftrag nicht ein weiteres Mal berechnet.

4.5.1. Initialisierung

Für die Initialisierung braucht der DistributionManager eine Vielzahl von Parametern. Dazu gehören

- der Name der verteilten Berechnung,
- die Anzahl der Teilaufträge,
- Referenzen auf Parameterdateien, den Anwendungscode und ein Verzeichnis, in dem Ergebnisse abgelegt werden sollen,
- (falls existent) die Beschreibungsdatei des ursprünglichen Auftrags,
- eine Referenz auf die Konfigurationsdatei,
- die Jobid,
- der Typ des Auftrags.

Die Angabe einer gültigen Referenz auf eine bereits existierende Beschreibungsdatei entscheidet über den Modus des Servers, also ob dies der Webserver der Initiators eines Auftrags sein soll oder nur ein weiterer Download-Server zum Load-Balancing, der nur den Anwendungscode weiterverteilt.

Im Root-Modus muss geprüft werden, ob die korrekte Anzahl an Parameterdateien vorliegt, d.h. ob die Anzahl der Parameterdateien mit der Anzahl der zu verschickenden Teilaufträge übereinstimmt. Sollte hier ein Fehler auftreten, so wird dieser Fehler zur aufrufenden Instanz propagiert, damit der Nutzer, der diesen Auftrag initiiert hat, benachrichtigt werden kann.

Im nächsten Schritt wird der Port des HTTP-Servers gebunden. Im Falle des Root-Modus wird nun eine neue Jobid, die URLs für Anwendungscode, Parameterdateien und das Ergebnisverzeichnis erzeugt und initialisiert. Im normalen Modus wird nur eine neue URL für den Anwendungscode erzeugt und anstelle der „alten“ URL in die Beschreibungsdatei für den Auftrag eingetragen. In beiden Fällen enthält eine URL immer die Jobid des aktuellen Auftrags.

Bei der initialen Erzeugung des Wurzel-WorkSpecFiles eines Auftrags wird die URL des Anwendungscodes noch in einem zweiten Feld gespeichert, welches niemals überschrieben wird. Dieses Feld dient dazu, dass jeder an der Berechnung teilnehmende

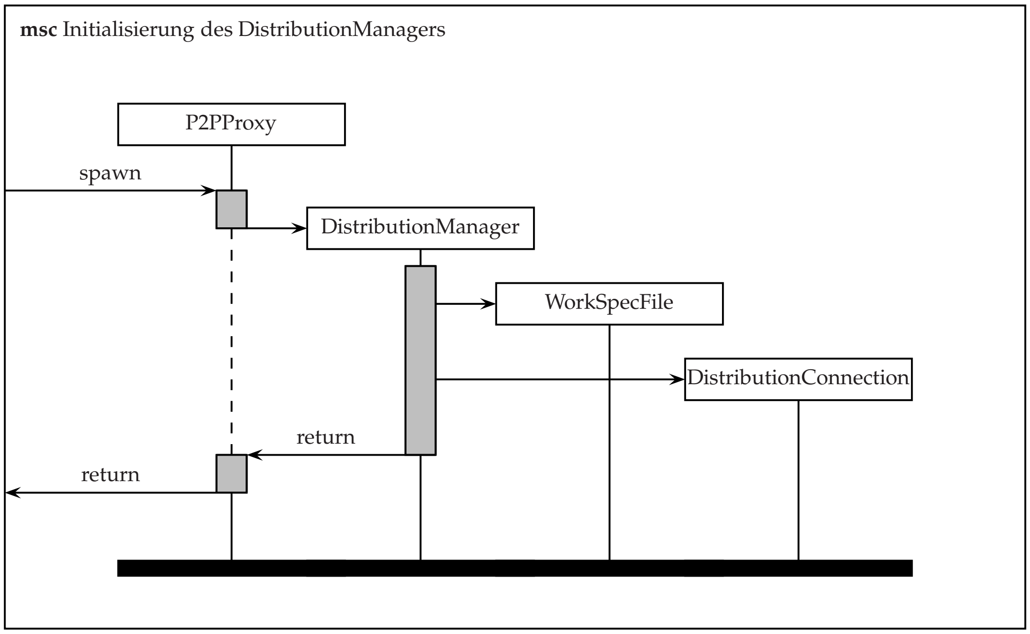

Abbildung 4.5.: Initialisierung des DistributionManagers

Knoten auf jeden Fall eine funktionierende URL hat, bei der der Anwendungscode heruntergeladen werden kann. Alle anderen Download-Quellen eines Auftrags müssen nicht zwangsweise bis zum Ende einer Berechnung gültig sein.

Die eigentliche Verarbeitung der unterschiedlichen Anfragen an den DistributionManager erfolgt in einer konfigurierbaren Anzahl von Threads der Klasse *DistributionConnection*. Initial warten diese auf eingehende Verbindungen. Die konkrete Verarbeitung von Anfragen ist im Folgenden beschrieben, wobei andere Anfragen außer PUT und GET direkt verworfen werden, d.h. mit der Meldung *HTTP/1.0 400 Bad Request* abgelehnt werden.

Die Initialisierung des DistributionManagers durch den P2PProxy wird in Abbildung 4.5 verdeutlicht.

4.5.2. Generelle Anfrageverarbeitung

Generell wird bei Eintreffen einer neuen Verbindung auf dem Eingangsstrom ein *DataInputStream* geöffnet mit dem für die HTTP-Header-Zeilen der eingehende Bytestrom zeilenweise, d.h. jeweils bis zum nächsten Zeilenende, der gesamte Header gelesen wird. Anschließend kann entsprechend der Request-Methode und der URL entschieden werden, welcher Code ausgeführt wird.

Die Funktionalität des Zeilenlesens wird eigentlich bereits von der bestehenden Klasse *InputStreamReader* bzw. vom *BufferedInputStream* erbracht. Allerdings hat bereits der

InputStreamReader den Nachteil, dass er Daten vorpuffert, und damit mehr Daten aus dem eingehenden Bytestrom liest, als er im Moment benötigt. So wird also schon über das Zeilenende hinaus gelesen, obwohl dort vielleicht gar keine Zeichen, sondern nur noch uninterpretierte Bytes folgen (Berechnungs-Ergebnisse in serialisierter Form). Deshalb wurde eine eigene Methode zum Lesen von Binärdaten bis zum Zeilenende und Interpretieren dieser Binärdaten als String realisiert.

4.5.3. Herunterladen von Parametern oder Anwendungscode

Liegt ein HTTP-GET-Request vor, so sollen entweder Anwendungscode oder Parameter heruntergeladen werden. Die Unterscheidung wird anhand der angefragten Ressource vorgenommen. In beiden Fällen muss die angefragte URL mit der lokal gespeicherten URL, für die dieser DistributionManager zuständig ist, übereinstimmen. Ist dies nicht der Fall, so wird der Status *HTTP/1.0 404 Not Found* gesendet.

Beim Download von Code wird der Content-Type *application/x-java-archive* gesetzt, eine korrekte Content-Length übergeben und dann als HTTP-Response-Body die binäre Jar-Datei in den Stream gelegt.

Beim Download von Parametern wird aus der angefragten Ressource herausgelöst, welche Parameterdatei genau angefragt wird. Sollte diese Parameterdatei noch zur Verfügung stehen, d.h. das Ergebnis der Berechnung dieser Parameterdatei liegt noch nicht vor, so wird die Parameterdatei als Content-Type *application/octet-stream* mit der passenden Content-Length in den Ausgabestream gelegt. Wenn die Parameterdatei nicht mehr berechnet werden muss oder niemals existiert hat, so wird der Status *HTTP/1.0 410 Gone* gesendet.

Ein erfolgreicher Download wird immer mit dem Status *HTTP/1.0 200 OK* beantwortet.

4.5.4. Hochladen von Ergebnissen

Wenn ein PUT-Request vorliegt, sollen Ergebnisse an den DistributionManager übertragen werden. Auch hier muss der in der URL übersandte Pfad mit dem Pfad übereinstimmen, den der DistributionManager vorgibt. Das bedeutet vor allem, dass die Jobid übereinstimmen muss. Ist dies nicht der Fall, wird der Upload mit dem Status *HTTP/1.0 404 Not Found* abgebrochen.

Die URL enthält auch die Id des Teilauftrags, für den ein Ergebnis übermittelt werden soll. Damit können eventuell bestehende Parameterdateien invalidiert werden. Aus der Angabe der Content-Length im Header des Requests wird ermittelt, wie viele Bytes das zu übermittelnde Ergebnis haben wird. Daraufhin wird versucht, nach Ende des Headers diese Anzahl an Bytes zu lesen. Die so empfangenen Daten werden dann gespeichert und eine eventuell zugehörige Parameterdatei wird invalidiert. Ansonsten wird der Zähler der noch benötigten Resultate dekrementiert.

Ein erfolgreicher Upload wird gemäß des HTTP-Standards [FGM+99] mit dem Status *HTTP/1.0 204 No Content* beantwortet. Tritt ein Fehler beim Upload auf, so wird dem Client der Status *HTTP/1.0 500 Internal Server Error* geschickt. Dies kann z.B. passieren, wenn beim Lesen des Streams ein Lesefehler auftritt.

4.6. P2PMessage

Die P2PMessage ist der Nachrichtentyp, der zwischen zwei Instanzen der Klasse *P2P-Proxy* verschickt wird. Eine Nachricht dieses Typs kapselt Sender- und Empfängerinformationen, den Typ der Nachricht, eine fortlaufende Sequenznummer sowie typspezifische Binärdaten.

Zur Vereinfachung für die P2P-Schicht sind einige Nachrichtentypen vorgesehen, die nicht zum Betrieb der Proxy-Ebene benötigt werden, sondern allgemein benötigte Nachrichten für P2P-Protokolle sind. Das Empfangen einer Nachricht dieses Typs löst direkt einen Methodenaufruf in der P2P-Schicht aus, statt eine Nachricht an die höher liegende Schicht auszuliefern.

4.6.1. Nachrichtentypen

JOIN: Beitrittswunsch zu einem P2P-Netz.

ACCEPT: Antwort auf JOIN. Beitritt zum Netz akzeptiert.

REDIRECT: Antwort auf JOIN. Beitritt an diesem Knoten abgelehnt. Umleitung zu anderem Knoten (z.B. wegen Überlast).

LEAVE: Benachrichtigung über kontrolliertes Beenden eines Knoten.

FINISHED: Benachrichtigung, dass ein Job vollständig bearbeitet ist.

SEND_CONFIRMED: Generische Nachricht, deren Empfang bestätigt werden muss.

SEND_UNCONFIRMED: Generische Nachricht, deren Empfang nicht bestätigt werden muss.

ACK: Quittung für eine generische Nachricht.

JOB_ANNOUNCE: Nachricht zum Verbreiten eines Auftrages.

JOB_ACCEPT: Nachricht, dass ein Auftrag akzeptiert wird.

4.7. EpidemicProtocol

Das *EpidemicProtocol* ist die Realisierung eines P2P-Protokolls, das auf epidemischen Algorithmen beruht. Nachrichten werden in regelmäßigen Abständen zwischen Knoten ausgetauscht, wobei auch regelmäßig Nachbarschaftslisten (auch „Host-Listen" genannt) ausgetauscht werden. Dieser regelmäßige Austausch von Nachbarschaftslisten (welche in Form der Klasse *PeerList* implementiert sind), erfolgt in jedem Ausführungszyklus mit einem zufällig ausgewählten Knoten aus der Liste. Diesem Peer wird nicht nur die eigene Liste geschickt, sondern dieser antwortet, indem er seiner eigene Liste zurückschickt. Durch diesen Handshake wird auch gleichzeitig die Round-Trip-Time zu diesem Knoten gemessen.

Das aktive Senden von Nachrichten erfolgt immer nur zu festen, regelmäßigen Sendezeiten; Nachrichten der Anwendungsschicht werden so lange in eine Warteschlange eingereiht. Reaktionen auf eingehende Nachrichten, z.B. die Antwort auf eine eingehende Nachbarschaftsliste oder die Antwort auf das Angebot eines Arbeitsauftrags, werden sofort versandt, da der Absender an dieser Stelle nur eine begrenzte Zeit auf eine Antwort wartet.

4.7.1. Adressierung

Innerhalb der P2P-Schicht ist die Adresse eines anderen Knotens die Uid – eine 64-bit-Zahl, deren Abbildung auf die in der Proxy-Schicht verwendete NetworkId durch die PeerList (siehe Abschnitt 4.7.4) erfolgt. Schichten oberhalb der P2P-Schicht können somit von der konkreten IP-Adresse der Knoten abstrahieren. Diese kann sich potentiell auch während der Lebenszeit des Knoten ändern. So können auch dynamische Netzwerkadressen unterstützt werden.

4.7.2. EpidemicMessage

Die *EpidemicMessage* ist die Nachrichtenklasse, die zwischen zwei Instanzen des EpidemicProtocol ausgetauscht wird. Epidemische Nachrichten dienen der Kommunikation zwischen zwei P2P-Instanzen zum Austausch von Nachbarschaftslisten, Arbeitsaufträgen oder Nachrichten zwischen verteilten Berechnungen. Der Versand von epidemischen Nachrichten erfolgt durch Serialisierung zu einem Byte-Array und Übergabe dieses Arrays an die gewünschte Sendefunktion der Proxy-Schicht.

Die EpidemicMessage dient auch wieder zum Kapseln von mehreren Informationen, die mit einer Nachricht von einem Knoten an einen anderen verbunden sind. Dazu gehören wiederum Netzwerkadresse von Sender und Empfänger, Typ der Nachricht, Uid und Jobid des aktuellen Auftrags des sendenden Knoten sowie zusätzliche typspezifische binäre Daten.

Serialisierung

Die Klasse EpidemicMessage besitzt eine Methode zum Serialisieren und einen Konstruktor zum Deserialisieren von Nachrichten. Die Serialisierung erfolgt aus Effizienzgründen wie bei vielen anderen Objekten dieser Implementierung nicht mit den Java-eigenen Methoden zur Serialisierung. Da diese serialisierten Formen noch zusätzliche Sicherungsdaten enthalten (z.B. den Typ des serialisierten Objektes) werden Instanzen der EpidemicMessage durch eine *toByteArray()*-Methode serialisiert. Diese Methode konvertiert die nötigen Daten in einen Byte-Strom. So werden z.B. in den EpidemicMessage-Objekten die NetworkIds von Sender bzw. Empfänger gehalten. Diese werden allerdings nicht serialisiert bzw. deserialisiert, da diese Informationen bereits auf der Proxy-Ebene mitgeschickt werden. Sie sind dennoch im Nachrichtenobjekt enthalten, damit man an einer zentralen Stelle auf alle nötigen Daten zugreifen kann.

Das Deserialisieren erfolgt durch Übergeben von Absender und serialisierter Nachricht an den Konstruktor einer EpidemicMessage. Dieser setzt direkt den übergebenen Absender in das Objekt ein und liest die restlichen Werte aus dem erhaltenen Byte-Array aus.

Nachrichtentypen

HOSTLIST: Aktives Verschicken einer Nachbarschaftsliste.

HOSTLIST_REPLY: Verschicken einer Nachbarschaftsliste als Antwort auf den Erhalt einer Nachbarschaftsliste.

JOB_ANNOUNCE: Nachricht zum Verbreiten eines Auftrags.

JOB_TAKEN: Quittung, dass ein Auftrag akzeptiert wurde und berechnet wird.

CHORD_ANNOUNCE: Besondere Nachricht zum beschleunigten initialen Versenden eines Auftrags, siehe auch Abschnitt 3.3.

DATA_UNCONFIRMED: Verschicken von unbestätigten Nachrichten ohne weitere Bedeutung für das System (z.B. durch eine verteilte Anwendung).

DATA_CONFIRMED: Verschicken von bestätigten Nachrichten ohne weitere Bedeutung für das System.

4.7.3. Abstraktion „Knoten“

Die Repräsentation eines einzelnen Knoten im implementierten P2P-Protokoll sind Objekte der Klasse *Peer*. Ein *Peer* ist ein Container für alle Daten, die um einen Knoten herum wichtig sind. Dazu gehören:

NetworkId: Seine Id auf Netzwerkebene bzw. im Middleware-Kontext die Id auf Proxy-Ebene, aus der sich direkt seine Adresse im IP-Netz ableiten lässt.

Uid: Eine eindeutige Id aus einem 64-bit-Adressraum, welche beim Initialisieren des System zufällig gleichverteilt gewählt wird.

Jobid: Eine eindeutige Id des Auftrags, an dem dieser Knoten gerade rechnet. Ein Wert von -1 bedeutet, dass dieser Knoten gerade „frei" ist.

Alter: Der Zeitpunkt, wann ein Knoten zum letzten Mal eine Nachricht von diesem Knoten erhalten hat, wobei hierfür kein globales Wissen zugrunde gelegt wird.

Job-Alter: Der Zeitpunkt, wann die Jobid des Knotens zum letzten Mal aktualisiert wurde.

RTT: Die Round-Trip-Time (Zeitbedarf einer Nachricht zu diesem Knoten und zurück). Ein Wert von -1 bedeutet, dass die Round-Trip-Time zu diesem Knoten unbekannt ist. Die RTT ist nur lokal von Bedeutung.

Serialisierung

Gerade die Objekte der Klasse Peer müssen sehr effizient serialisiert werden, da in einer Nachricht mit einer Nachbarschaftsliste viele Peers enthalten sein können. Damit wirkt sich eine Ersparnis bei der Serialisierung von Peers am deutlichsten aus.

Aus diesem Grund werden auch Peers „manuell" serialisiert, denn auch hier ergeben sich nicht nur Einsparungen dadurch, dass ein Serialisierungs-Header mit Klasseninformationen weggelassen wird, sondern ein Peer-Objekt enthält Daten, die gar nicht verschickt werden müssen, da sie nur lokale Bedeutung haben. Ein Verschicken der Round-Trip-Time macht keinerlei Sinn, da diese Daten aus Sicht eines anderen Knoten vollkommen anders aussehen können.

Ein besonderes Augenmerk wird auch auf die Alters-Informationen gelegt. Beim Serialisieren dieser beiden Felder macht es keinen Sinn, die absolute Zeit in binärer Form zu übertragen, da unterschiedlich gehende Systemuhren Verfälschungen ergeben würden. Vielmehr werden bei der Übertragung nur die Anzahl der Sekunden seit der letzten Nachricht bzw. der letzten Änderung der Jobid übertragen, so gibt es nur eine sehr geringe Abweichung, verursacht durch den Zeitraum zwischen Serialisierung und Deserialisierung.

4.7.4. Nachbarschaftsliste

Nachbarschaftslisten, die durch das epidemische Protokoll ausgetauscht werden, sind durch die Klasse *PeerList* implementiert, welche in der Basis aus einer Menge von Peers besteht. Da die Nachbarschaftslisten regelmäßig über das Netz ausgetauscht werden, benötigt die Liste eine Funktionalität, Listen miteinander mischen zu können. Zu diesem Zweck existiert eine *merge()*-Methode, welche alle Knoten aus der erhaltenen Liste in die eigene einfügt und anschließend dafür sorgt, das die Nachbarschaftsliste eine als maximal definierte Länge nicht überschreitet. Diese Längenbeschränkung der PeerList wird in der Konfigurationsdatei der Middleware definiert und dient der Skalierung des P2P-Netzes, um den Speicherbedarf zu begrenzen.

Knoten, die ein maximales Alter überschreiten, werden in der Liste gar nicht erst eingefügt. In der aktuellen Implementierung ist dieses Alter auf fünf Minuten festgesetzt, d.h. ein Knoten, von dem seit über fünf Minuten keine Nachricht mehr empfangen wurde, wird beim Mischen der Listen nicht mehr als lebendig betrachtet.

Für die Realisierung einer beschleunigten Nachrichtenverteilung gemäß eines Chord-ähnlichen Algorithmus (siehe Abschnitt 3.3) ist eine *Map* implementiert, welche von Adressen innerhalb des Chord-Adressraums auf Peer-Objekte abbildet.

Hinzufügen von Knoten

Beim Hinzufügen von Knoten muss darauf geachtet werden, dass ein Knoten bereits in der Liste enthalten sein kann. In diesem Fall muss geprüft werden, ob das Alter das empfangenen Knotens größer oder kleiner als das Alter des Peers in der lokalen Liste ist. Sowohl beim Alter des Knoten selbst als auch beim Alter der Jobid muss der jüngere Eintrag übernommen werden. Für den Fall, dass sich die Uid eines Knotens geändert hat (was dadurch vorkommen kann, dass eine Middleware-Instanz beendet und mit der gleichen Netzwerk-Adresse neu gestartet wurde), wird die neue Uid übernommen und die Chord-Liste neu berechnet.

Suchen von Knoten

Ein Knoten kann in der Liste nach mehreren Kriterien gesucht werden. Knoten können mit sechs verschiedenen Anfragen gefunden werden:

NetworkId: Der Knoten mit dieser NetworkId wird zurückgeliefert.

Uid: Der Knoten mit dieser Uid wird zurückgeliefert.

ChordId: Hier wird derjenige Knoten, der im strukturierten Netz diese Adresse besitzt, zurückgegeben.

Jobid: In diesem Fall wird eine Liste von Knoten zurückgeliefert, da meist mehrere Knoten am gleichen Auftrag rechnen.

zufällig: In diesem Fall wird ein zufälliger Knoten zurückgeliefert.

zufällig ohne Job: Bei dieser Methode wird ein zufälliger Knoten geliefert, der gerade keinen Auftrag hat (d.h. dessen Jobid -1 ist).

Wenn kein zur Anfrage passender Knoten gefunden wird, so wird ein leeres Objekt (*null*) zurückgegeben.

Serialisierung

Die PeerList benutzt zwar den Java-eigenen Mechanismus zur Serialisierung, überschreibt aber die Methoden, wie die Serialisierung im Detail erfolgt. So werden nicht alle Felder serialisiert, da es nicht nötig ist, diese mit anderen Knoten auszutauschen. Dazu werden die Methoden *readObject* und *writeObject* überschrieben. Nach dem Header, den Java standardmäßig bei der Serialisierung schreibt, wird nur noch die Anzahl der übertragenen Knoten und danach die einzelnen Knoten in serialisierter Form geschrieben.

4.7.5. CyclicSender

Der *CyclicSender* erbt von der Klasse *Thread* und ist im Rahmen des epidemischen Protokolls dafür zuständig, regelmäßig Nachrichten an andere Knoten zu verschicken. Der CyclicSender läuft – bis er von außen abgebrochen wird – in einer Endlosschleife.

Verschicken von Nachbarschaftslisten

Bei jedem Schleifendurchlauf wird, sofern kein Abbruchwunsch festgestellt wird, die Nachbarschaftsliste an einen beliebigen anderen Knoten verschickt und auf dessen Antwort gewartet (oder bis ein Timeout festgestellt wird). Wenn der Thread nicht rechtzeitig antwortet, wird er aus der Nachbarschaftsliste gestrichen. Die Rechtzeitigkeit ist in der Konfigurationsdatei definiert. Bei rechtzeitiger Antwort wird die Round-Trip-Time entsprechend aktualisiert.

Verschicken von Arbeitsaufträgen

Nach dem Versenden der Nachbarschaftsliste wird – sofern vorhanden – ein aktuell zu bearbeitender Arbeitsauftrag verschickt. Dazu wird aus der Nachbarschaftsliste ein zufälliger Knoten bestimmt, der gemäß dem aktuellen Kenntnisstand gerade „frei“ ist. Wenn kein solcher Knoten gefunden wird, so wird in dieser Runde kein Arbeitspaket verschickt. Ansonsten wird der aktuelle Arbeitsauftrag in zwei Teile geteilt, die eine Hälfte verschickt und die andere Hälfte gemerkt. Das „Merken“ erfolgt deshalb, weil der andere Knoten erst bestätigen muss, dass er das erhaltene Arbeitspaket auch akzeptiert und verarbeitet. Sollte diese Bestätigung ausbleiben, so wird in der nächsten

Runde das gleiche Arbeitspaket an einen anderen zufällig bestimmten Peer versendet. Wenn die Bestätigung eintrifft, so wird das „gemerkte" Arbeitspakete als das aktuelle angesehen und in der nächsten Runde die Hälfte davon versendet.

Es ist noch hervorzuheben, dass alle n Runden nicht das aktuelle lokale Arbeitspaket verschickt wird, sondern – sofern vorhanden – die Beschreibungsdatei eines lokal gestarteten Auftrags, in der genau vermerkt ist, welche Teilaufträge noch fehlen. Der Wert von n kann in der Konfigurationsdatei festgelegt werden. Bei Bedarf kann diese Funktion auch abgeschaltet werden, es ist dann jedoch möglich, dass durch Knotenausfälle oder verlorene Nachrichten Teilaufträge niemals eintreffen. Nichtsdestotrotz wird ein Auftrag immer irgendwann beendet, im Zweifelsfall dadurch, dass alle ausstehenden Teilaufträge lokal berechnet wurden.

Das Versenden eines Arbeitspaketes erfolgt immer als unbestätigte Nachricht, kann also auch jederzeit verloren gehen.

Verschicken von Monitor-Nachrichten

Zu Kontrollzwecken kann der CyclicSender auch Kontrollnachrichten an einen zentralen Monitor versenden. Da dieser zentrale Monitor ein Flaschenhals ist, ist er nicht für den Normalbetrieb gedacht und auch nicht, um von allen Knoten Nachrichten zu erhalten. Der Monitor kann durch erhaltene Nachrichten Rückschlüsse auf eine mögliche aktuelle Topologie des P2P-Netzes ziehen und einen Graphen mit Nachbarschaftsbeziehungen zeichnen. Außerdem können an ihn Nachrichten verschickt werden, an welchem Auftrag ein Knoten gerade rechnet.

Verschicken von Anwendungsnachrichten

In epidemischen Protokollen ist es üblich, dass nicht jederzeit Nachrichten versandt werden, sondern nur zu bestimmten Zeiten. Dies gilt auch für die Nachrichten auf Anwendungsebene. Wenn die Anwendung Nachrichten verschicken will, so werden diese Nachrichten in eine Warteschlange gestellt und erst zum nächsten Sendezeitpunkt, d.h. beim nächsten Schleifendurchlauf des CyclicSender versandt. Dies erfolgt im Anschluss an das Senden von Nachbarschaftsliste und aktuellem Arbeitsauftrag. Zu diesem Zeitpunkt werden alle in der Warteschlange befindlichen Nachrichten versandt, je nach Typ bestätigt oder unbestätigt.

Zykluszeit

Die Zykluszeit des CyclicSender ist im Prinzip durch die Konfigurationsdatei bestimmt. Allerdings wird dort nicht ein festes Raster definiert, an dem der Thread aufwacht und Nachrichten verschickt, sondern die Dauer, die der Thread zwischen zwei Ausführungen schläft. Die Zykluszeit schwankt also mit der Dauer der Ausführungszeit des

Threads. Im Idealfall werden Nachrichten in dem Takt verschickt, der durch die Konfigurationsdatei vorgegeben wird. Im Normalfall kommen der Zeitaufwand für das Schicken von Nachrichten, das Serialisieren von Nachbarschaftsliste und Arbeitsauftrag, das Bestimmen eines zufälligen Peers, das Teilen des Arbeitsauftrags, usw. dazu. Der variabelste Faktor ist jedoch die Zeitdauer, die auf die Antwort eines anderen Peers beim Austausch der Nachbarschaftslisten gewartet wird. Diese Zeit wird durch einen Timeoutwert begrenzt.

4.7.6. Netzbeitritt

Der Netzbeitritt wird durch den Aufruf der *connect()*-Methode ausgelöst. Zu diesem Zeitpunkt werden die nötigen Join-Parameter aus der Konfigurationsdatei ausgelesen und so lange im konfigurierten Zeitabstand Join-Nachrichten verschickt (durch Aufruf der *join()*-Methode der Proxy-Ebene), bis eine dieser Nachrichten beantwortet wird (positiv oder negativ) oder bis die maximale Anzahl der Nachrichten verschickt wurde. Sollte keine positive Antwort eingegangen sein, so wird die aufrufende Instanz durch eine Exception benachrichtigt, dass der Join nicht erfolgreich war. Der Standardfall ist in Abbildung 4.6 dargestellt.

Konzeptionell könnte es passieren, dass als Antwort auf eine Join-Nachricht nicht ein Accept gesendet wird, sondern der Knoten per Redirect-Nachricht zu einem anderen Knoten umgeleitet wird. Dieser Fall ist für das implementierte epidemische Protokoll allerdings nicht vorgesehen, es werden grundsätzlich nur Accept-Nachrichten verschickt. Der Empfang der Redirect-Nachricht ist also nur rudimentär, d.h. unter Ausgabe einer Fehlermeldung implementiert. Ein möglicher Nachrichtenverlust an dieser Stelle wurde daher nicht berücksichtigt.

Bei erfolgreichem Netzbeitritt werden der CyclicSender und der Thread, der empfangene Nachrichten bearbeitet, initialisiert und gestartet.

4.7.7. Starten von Aufträgen

Das Starten von Aufträgen ist ein recht komplexer Vorgang und wird durch den Aufruf der Methode *new_job()* ausgelöst. Das Starten von Aufträgen ist ein Vorgang, der potentiell von mehreren Threads angestoßen werden kann, zum Einen vom dem Thread, der Benutzerinteraktionen verarbeitet, zum Anderen von dem Thread, der eingehende Nachrichten bearbeitet. Da sich diese beiden Threads nicht gegenseitig behindern dürfen, wird das Starten von Aufträgen durch ein eigenes Objekt synchronisiert, so dass der Start von zwei Aufträgen nie gleichzeitig erfolgen kann.

Im nächsten Schritt wird geprüft, ob der lokale Knoten hinter einem Proxy-Server sitzt. Wenn es sich um einen Proxy-Client handelt, so macht es keinen Sinn, einen eigenen DistributionManager (siehe Abschnitt 4.5) zu starten, da kein Knoten darauf zugreifen kann, weil potentiell eine Firewall dies verhindert. Ansonsten wird zur Verteilung der

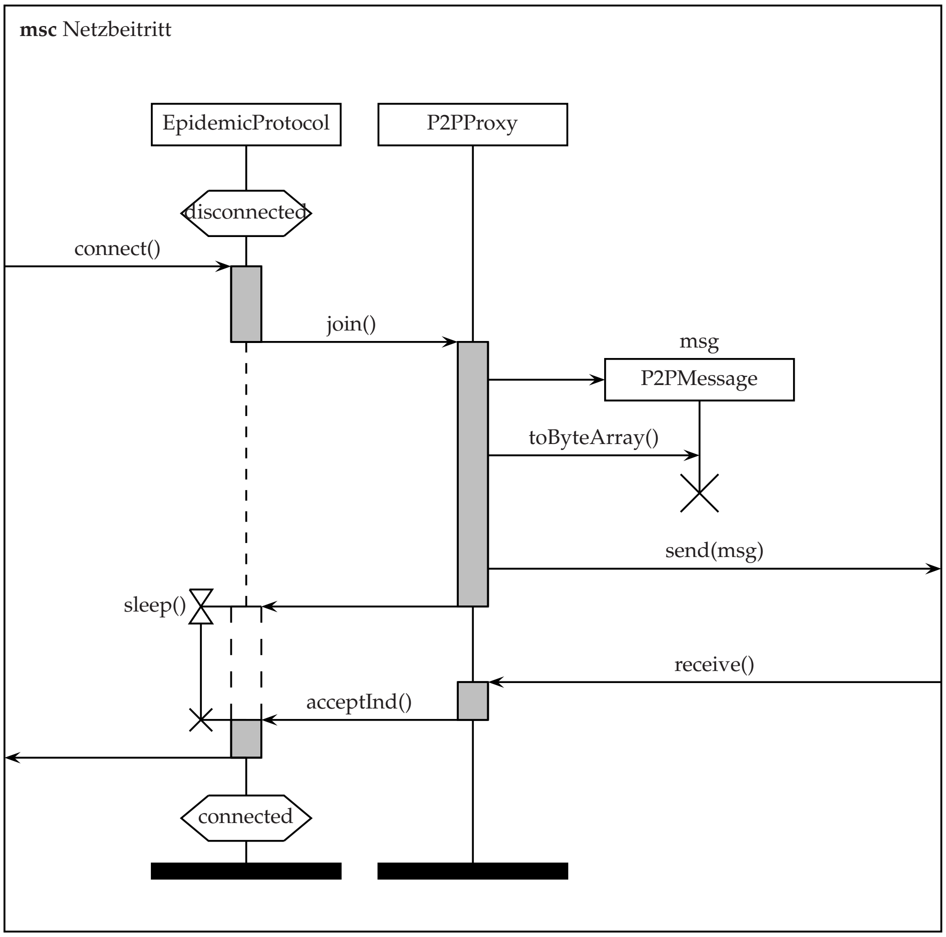

Abbildung 4.6.: Netzbeitritt

Download-Last ein eigener DistributionManager gestartet und das von diesem zurückgegebene WorkSpecFile weiterverwendet.

Während dieser Job initial verteilt wird, wird gegen die Beschreibungsdatei synchronisiert, damit der Auftrag nicht bereits zu früh durch den CyclicSender verteilt wird. Noch bevor der Auftrag an andere Knoten versendet wird, wird jedoch bereits begonnen, den Auftrag lokal zu bearbeiten.

Anschließend muss die Auftragsbeschreibung an andere Knoten verteilt werden, dazu wird initial ein Chord-ähnlicher Broadcast (siehe 3.3) eingesetzt, mit dessen Hilfe der Auftrag schnell und mit geringer Redundanz verteilt wird. Hierzu wird das Auftragspaket nacheinander gezielt an bestimmte Knoten der Nachbarschaftsliste verschickt. An dieser Stelle wäre es denkbar, auf eine Bestätigung dieser Knoten zu warten, um Paketverlust im Netzwerk zu maskieren, allerdings würde das den Zeitbedarf des Broadcasts deutlich verlängern. Durch regelmäßiges epidemisches Verschicken von Auftragspaketen ist der Verlust von Paketen durch das Chord-ähnliche Protokoll abgesichert.

Nachdem der Auftrag lokal gestartet wurde und das initiale Versenden der Auftragsbeschreibung abgeschlossen ist, ist der Auftragsstart abgeschlossen. Die weitere Berechnung von Teilaufträgen sowie das regelmäßige Versenden von ausstehenden Teilaufträgen wird von anderen Threads übernommen.

Sollte irgendwo im Rahmen der Startprozedur für einen Auftrag ein Fehler auftreten, z.B. dadurch, dass der DistributionManager nicht gestartet werden konnte oder dass ein Fehler beim Versenden von Nachrichten aufgetreten ist, so wird die Verteilung abgebrochen und ein lokaler Benutzer, falls er der Auftraggeber war, benachrichtigt. Die Benachrichtigung erfolgt dadurch, dass der Nutzer als Jobid seines neu gestarteten Auftrags den Wert -1 zurückgeliefert bekommt.

Einen besseren Eindruck vom Ablauf eines Auftragsstarts gibt Abbildung 4.7.

4.7.8. Empfang von Aufträgen

Der Empfang eines Arbeitsauftrags kann vom Systemdesign her durch die Proxy-Schicht erfolgen. In der vorliegenden Implementierung wurde der Versand von Auftragspaketen jedoch komplett innerhalb der P2P-Schicht implementiert, d.h. ein Auftrag wird durch Versand von Nachrichten auf Ebene der P2P-Schicht, also mittels einer *EpidemicMessage*, realisiert. Der Empfang eines Arbeitsauftrags wird hier also nicht durch den Empfangs-Thread der Proxy-Schicht, sondern durch den der P2P-Schicht gesteuert.

Wenn also ein Auftrag durch eine EpidemicMessage des Typs „JOB_ANNOUNCE“ empfangen wird, so wird als erstes gegen das bereits oben erwähnte Objekt zum Auftragsstart synchronisiert, damit nur ein Auftrag gleichzeitig initiiert werden kann.

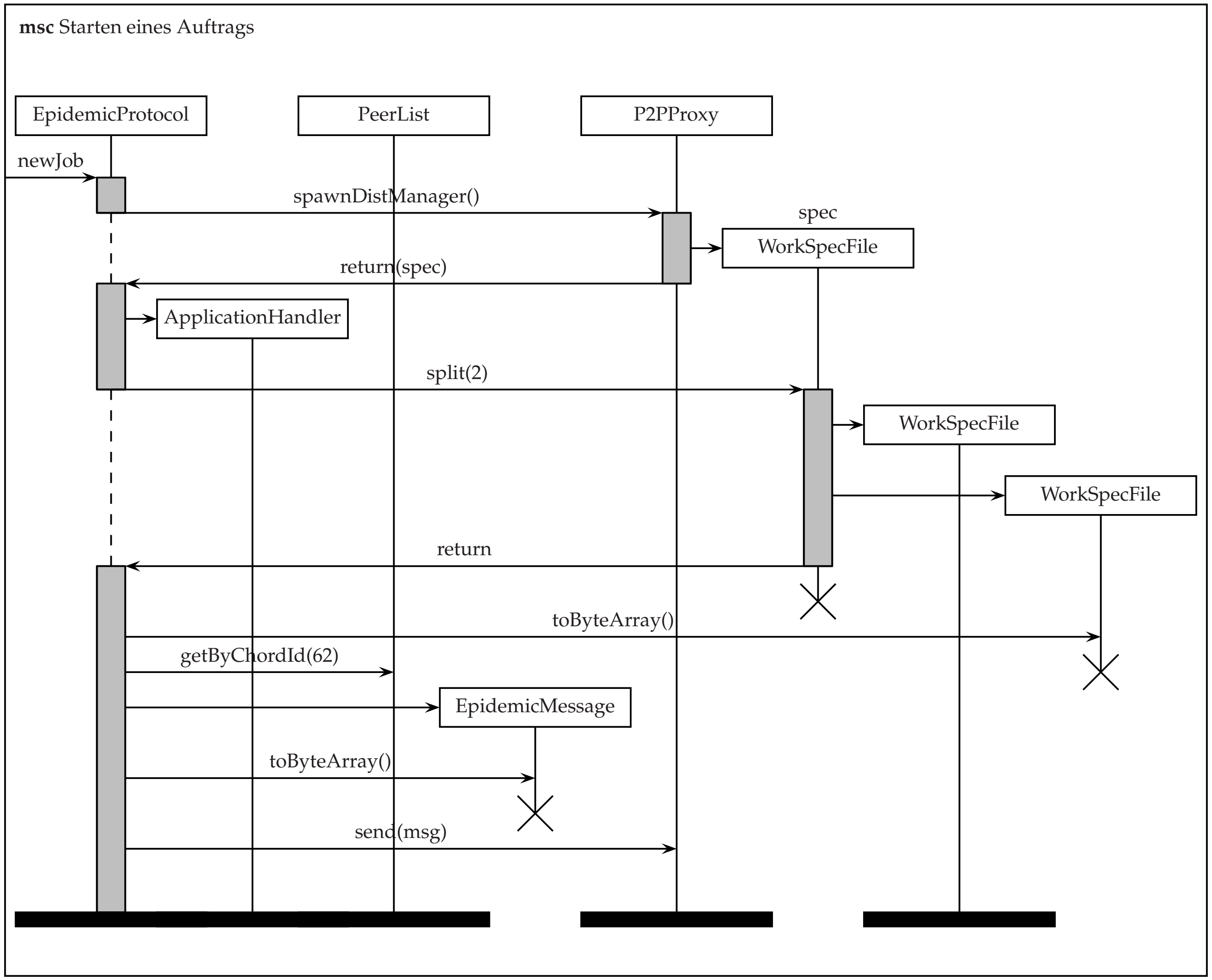

Abbildung 4.7.: Starten eines Auftrags

Akzeptieren von Aufträgen

An dieser Stelle kann geprüft werden, ob ein empfangener Auftrag überhaupt bearbeitet werden kann oder soll. Die Entscheidung, einen Auftrag zu akzeptieren, oder sogar einen anderen bereits laufen Auftrag dafür zu beenden oder auch zwei Aufträge parallel zu bearbeiten, kann von vielen Faktoren abhängen. In der vorliegenden Implementierung ist das einzige Kriterium jedoch, ob der lokale Knoten bereits an einem Auftrag rechnet. Ist der Knoten beschäftigt, so wird der Auftrag abgelehnt, anderenfalls wird der Auftrag akzeptiert. Das Ablehnen eines Auftrags erfolgt einfach, indem das Paket ignoriert wird. Bekommt der Absender keine Bestätigung, so wird er den Auftrag an einen anderen Knoten schicken.

Der Auftrag wird angenommen, indem sofort eine Quittung (d.h. ein Paket vom Typ „JOB_TAKEN“) verschickt wird. Inhalt des Paketes ist die Jobid des Auftrags, der akzeptiert wurde.

Verarbeiten eines empfangenen Auftrags

Wenn ein Arbeitsauftrag empfangen wird, wird als erstes geprüft, ob der Auftrag überhaupt gültig ist, d.h. ob überhaupt eine Beschreibungsdatei empfangen wurde. Sollte dies nicht der Fall sein, kann kein Auftrag gestartet werden und das Paket wird verworfen.

Es ist durchaus möglich, dass ein Knoten mehrfach einzelne Teilaufträge für den gleichen Gesamtauftrag bekommt. Es wäre hier ineffizient, wenn jedes Mal wieder der zugehörige Anwendungscode heruntergeladen werden muss. Außerdem ist es nicht denkbar, dass sich zwischen verschiedenen Teilaufträgen des selben Auftrags der Anwendungscode ändert. Daher wird an dieser Stelle ein Cache für Anwendungscode geführt, dessen Schlüssel die Jobids der Aufträge sind.

Sollte also ein Auftrag empfangen werden, so wird abhängig davon, ob der lokale Knoten hinter einem Proxy-Server sitzt und ob der Code bereits im Cache vorliegt, der Anwendungscode heruntergeladen, um diesen dann einem lokalen DistributionManager zu übergeben. Wenn der lokale Knoten ein Proxy-Client ist, so macht dies, wie oben erwähnt, keinen Sinn. Im Nicht-Proxy-Fall wird der Code jedoch heruntergeladen und in einem Cache zwischengespeichert oder einfach aus dem Cache geholt. Daraufhin wird wie im obigen Abschnitt „Starten von Aufträgen“ (siehe 4.7.7) der Auftrag gestartet.

4.7.9. Empfangen von Nachrichten

Das Verarbeiten empfangener Nachrichten erfolgt durch einen eigenen Thread, den *EpidemicWorker*. Dieser Thread liest blockierend auf einer Warteschlange für epidemische Nachrichten und verarbeitet diese dann. Bei jeder Nachricht, die von einem anderen Knoten empfangen wird, wird in der Nachbarschaftsliste festgehalten, dass dieser

Knoten soeben „gesehen“ worden ist, indem sein Alter auf die aktuelle Zeit in Millisekunden gesetzt wird. Die Verarbeitung unterscheidet sich je nach Typ der empfangenen Nachricht.

Nachbarschaftslisten

Der Empfang von Nachbarschaftslisten kann durch zwei verschiedene Nachrichtentypen geschehen. Zum Einen kann ein Knoten eine aktiv (durch den CyclicSender des fremden Knoten) gesendete Liste erhalten, zum Anderen kann er eine Liste als Antwort auf eine von ihm gesendete Nachbarschaftsliste erhalten. Die Verarbeitung dieser beiden Nachrichten erfolgt fast identisch, der Unterschied liegt nur darin, dass im ersten Fall eine Quittung, welche die eigene Liste enthält, verschickt wird. In beiden Fällen wird die empfangene Liste aus den empfangenen Binärdaten deserialisiert und dann mit der eigenen Liste gemischt. Vorher wird jedoch noch die Nachbarschaftsliste der aktuell berechneten Anwendung aktualisiert (siehe Abschnitt 4.8).

Generische Daten

Der Empfang von generischen Daten erfolgt hier ähnlich wie auch schon auf der Proxy-Ebene. Da diese Daten nur für die momentan ausgeführte verteilte Anwendung Bedeutung haben, werden diese Nachrichten an die entsprechende verteilte Anwendung zugestellt. Dies geschieht durch Zustellen der Nachricht beim betreffenden *Application-Handler*. Ist die Anwendung, für die diese Nachricht bestimmt ist, bereits terminiert, wird die Nachricht verworfen. Zur Sicherheit findet eine Überprüfung statt, ob die aktuell ausgeführte Anwendung tatsächlich mit der sendenden Anwendung identisch ist (durch Vergleich der Jobid). Sollte dies nicht der Fall sein, so wird die Nachricht ebenso verworfen.

Empfangen von Aufträgen

Das Empfangen von Arbeitsaufträgen wurde bereits oben in Abschnitt 4.7.8 beschrieben.

Empfangen von Auftragsquittungen

Das Verteilen von Aufträgen durch den CyclicSender beruht darauf, dass jeder Knoten durch den Versand einer Quittung signalisiert, dass er einen Auftrag auch akzeptiert. Der einzige Grund dafür, dass ein Knoten einen Auftrag nicht akzeptiert, ist (zumindest in der vorliegenden Implementierung), dass er bereits einen anderen Teilauftrag berechnet. Bei Empfang einer Quittung wird überprüft, ob der lokale Knoten gerade wirklich am Auftrag mit der in der Quittung mitgeschickten Jobid rechnet. Falls dies der Fall ist, so wird der Zeiger für das aktuelle Arbeitspaket auf das „gemerkte“ Arbeitspaket gesetzt (vgl. Abschnitt 4.7.5).

Empfang von Chord-Nachrichten

Durch den Chord-ähnlichen Algorithmus um Arbeitspakete initial schnell im Netz zu verbreiten werden beim Starten eines Auftrags Pakete gezielt an bestimmte Knoten im Netz verteilt. Beim Empfang eines solchen Paketes muss zum Einen, abhängig davon, ob der lokale Knoten bereits mit einer verteilten Berechnung beschäftigt ist, der Auftrag lokal gestartet werden. Auf jeden Fall muss der Auftrag aber an andere Knoten weitergeleitet werden. So wird sichergestellt, dass jeder Knoten im Netz (abgesehen von verlorenen Nachrichten) schnell über den Arbeitsauftrag informiert wird.

Der Verlauf wird mit einer Fallunterscheidung schnell deutlich:

- Wenn lokal „frei“, dann:
  - Wenn mehr als ein Teilpaket, dann teile den Auftrag, starte erste Hälfte lokal und verschicke die zweite Hälfte,
  - ansonsten starte den Auftrag nur lokal.
- ansonsten verschicke den ganzen Auftrag weiter.

Zu Kontrollzwecken für den Algorithmus werden jeweils bei jeder der Möglichkeiten entsprechende Nachrichten an den zentralen Monitor verschickt, sofern dies konfiguriert wurde.

4.7.10. Beenden von Berechnungen

Die P2P-Ebene stellt eine Methode bereit, mit der die Anwendungsebene der P2P-Ebene mitteilen kann, dass ein Teilauftrag fertig berechnet wurde. Diese Benachrichtigung dient dazu, eine erneute Verfügbarkeit der Ressourcen mitzuteilen.

Erfolgt diese Benachrichtigung noch während der Startphase eines Auftrags, dann kann die P2P-Schicht daraus schließen, dass ein Fehler während der Bearbeitung aufgetreten ist, also trotz Start einer Anwendung aktuell kein Teilauftrag bearbeitet wird. Eine mögliche Ursache an dieser Stelle ist z.B., dass der Anwendungscode nicht heruntergeladen werden konnte, ein Starten des nächsten Teilauftrags hätte hier also keine Sinn.

Ansonsten wird von einer normalen Fertigstellung eines Teilauftrags ausgegangen, so dass der nächste Teilauftrag berechnet werden kann, sofern noch einer vorhanden ist. Wenn kein „normaler“ Teilauftrag mehr vorhanden ist besteht noch die Möglichkeit, dass im Wurzelpaket noch Teilaufträge sind, die zwar eventuell verteilt wurden, aber noch nicht wieder abgeliefert wurden. Da der lokale Knoten ja keinen anderweitigen Auftrag hat, kann er an diesen Paketen mitrechnen, immerhin ist es ja der Auftrag des lokalen Nutzers.

Ein automatischer erneuter Start von weiteren Teilaufträgen aus dem aktuellen Arbeitspaket oder dem lokalen Wurzelpaket erfolgt so lange, bis weder normale Arbeitspakete noch Wurzelpakete vorhanden sind. Wenn keine Arbeit mehr vorhanden ist, beschränkt sich die Middleware wieder auf das Austauschen von Nachbarschaftslisten und wartet darauf, dass ein neuer Auftrag – entweder über das P2P-Netz oder vom lokalen Benutzer – gestartet wird.

4.8. ApplicationHandler

Der *ApplicationHandler* ist eine Implementierung der Anwendungsschicht (siehe Abschnitt 3.4.4). Instanzen dieser Klasse werden erzeugt, wenn die P2P-Schicht einen Arbeitsauftrag erhält und diesen lokal berechnen lässt. Der ApplicationHandler dient dazu, eine verteilte Anwendung zu starten, dieser eine Schnittstelle für das Versenden und Empfangen von Nachrichten zu bieten, sowie diese zu Überwachen.

4.8.1. Nachrichtendienste

Als Basisdienst wird einer verteilten Anwendung angeboten, beliebige Nachrichten in Form von Byte-Arrays an einen Knoten mit beliebiger Uid verschicken zu können. Gültige Uids von Knoten, die an der gleichen verteilten Berechnung teilnehmen, bekommt die verteilte Abwendung durch Aufrufen der Methode *getNeighbours()*. Zusätzlich zu diesem Basisdienst kann eine verteilte Berechnung noch ganze Objekte (inklusive anhängender Objektbäume) über die Middleware verschicken. Der Empfänger bekommt diese Daten dann über die entsprechende *receive()*-Methode zugestellt.

Neben der Möglichkeit, Nachrichten an einzelne Knoten zu verschicken, kann eine Instanz einer verteilten Berechnung auch Broadcasts, also eine Nachricht an alle Teilnehmer desselben Auftrags, sowie Multicasts verschicken. Beim Multicast kann die Anwendung als Parameter angeben, an wie viele andere Knoten die Nachricht verschickt werden soll. Sofern so viele andere Teilnehmer der Berechnung bekannt sind, wird daraufhin diese Anzahl an Nachrichten verschickt, ansonsten an alle bekannten.

Der Versand von Broadcast- und Multicast-Nachrichten erfolgt in der vorliegenden Implementierung ineffizient, d.h. durch Versand einzelner Unicast-Nachrichten an die Empfänger.

4.8.2. Überwachung der Anwendung

Sollte eine verteile Anwendung während der Berechnung abstürzen und kann den ApplicationHandler nicht mehr über das Ende der Berechnung benachrichtigen, so kann dieser feststellen, dass die Anwendung tot ist (bzw. dass der ausführende Thread nicht mehr existiert) und diese für beendet erklären, so dass die P2P-Schicht wieder neue Aufträge annehmen und bearbeiten kann.

Es ist ebenso denkbar, dass eine verteilte Berechnung außer Kontrolle gerät, und beispielsweise in eine Endlosschleife gerät. In diesem Fall wäre die Anwendung nicht tot, würde aber ewig Ressourcen verbrauchen und den lokalen Knoten für Ausführungen anderer Programme blockieren. Für diesen Fall kann eine maximale Ausführungszeit (in Echtzeit, nicht in CPU-Sekunden) konfiguriert werden, nach Ablauf derer die Anwendung vom ApplicationHandler mit Gewalt beendet wird. Diese Möglichkeit wird als wichtig erachtet, konnte jedoch nur durch die Benutzung der als „*deprecated*“ eingestuften Methode *Thread.stop()* realisiert werden. Ein Problem dieser Art und Weise, einen Thread zu beenden, ist nämlich, dass von diesem Thread gehaltene Synchronisationssperren eventuell nicht mehr freigegeben werden.

4.8.3. ApplicationMessage

Die *ApplicationMessage* ist ein Objekt zur Kapselung von Nachrichten auf Anwendungsebene. Da auf dieser Ebene keinen Nachrichtentypen mehr unterschieden werden, ist die ApplicationMessage nur ein Container, um die Daten der Nachricht selber und den Empfänger zu kapseln.

Die ApplicationMessage hat eine leicht andere Rolle als die P2PMessage und die EpidemicMessage. Sie dient nicht dazu, als serialisierte Form verschickt zu werden, wie dies bei den anderen beiden der Fall ist. Die ApplicationMessage wird erst beim Empfang im ApplicationHandler erzeugt, um den von der P2P-Schicht übertragenen Absender der Nachricht in der Eingangswarteschlange der Anwendungsschicht mit speichern zu können. An dieser Stelle ist auch nochmal erwähnenswert, dass hier nur die Uid des Absenders gespeichert wird, nicht die eigentliche NetworkId.

4.8.4. Starten eines Auftrags

Eine verteilte Berechnung wird bereits durch den Konstruktor des ApplicationHandler gestartet. Dort werden, sofern durch den Typ der Beschreibungsdatei des Auftrags vorgegeben, schon eventuell vorhandene Parameterdateien heruntergeladen, sowie die Nachbarschaftsliste für an diesem Auftrag teilnehmende Knoten und die Warteschlange für eingehende Nachrichten initialisiert und der Thread zum Abarbeiten der empfangenen Nachrichten gestartet. Wenn die Parameter für den aktuellen Teilauftrag nicht heruntergeladen werden können, so wird dieser Teilauftrag verworfen, da er vermutlich schon berechnet wurde. Bereits nach dem Laden der Parameterdaten werden diese Daten lokal invalidiert, da entweder die Berechnung zu einem erfolgreichen Ende kommt (und damit der Teilauftrag nicht nochmal berechnet werden muss) oder ein Fehler auftritt, der mit diesem Teilauftrag zusammenhängt (z.B. weil die Parameter für diesen Teilauftrag schon zentral beim Initiator invalidiert wurden). Bei Fehlern beim Herunterladen der Parameter wird nicht gleich der ganze Auftrag abgebrochen, sondern so lange versucht, den nächsten Teilauftrag herunterzuladen und diesen zu invalidieren, bis alle abgearbeitet wurden.

Anschließend wird der benötigte Anwendungscode vom im WorkSpecFile definierten Ort heruntergeladen, meist ist dies der lokale DistributionManager. Dies ist nur dann nicht der Fall, wenn sich der Knoten hinter einem Proxy-Server befindet, dann muss der Code direkt vom Absender des Teilauftrags geholt werden. Für den Fall, dass dieser die Berechnung schon beendet hat und der Anwendungscode dort nicht mehr heruntergeladen werden kann, ist als Sicherungslösung noch eine zweite URL in der Beschreibungsdatei enthalten, die die Quelle direkt beim Initiator des Auftrags angibt. Diese Quelle sollte aus Lastverteilungsgründen erst als letzter Ausweg gewählt werden.

Aus der Jar-Datei wird durch das Attribut *Main-Class* herausgelesen, welche Klasse aus der Jar-Datei zum Starten der verteilten Berechnung benutzt werden soll. Für diese wird dann ein eigener Thread erstellt, der die Methode *start()* auf einer Instanz dieser Klasse aufruft. Desweiteren werden Timer gestartet, die den Thread wie oben erwähnt (siehe Abschnitt 4.8.2) überwachen.

Sollte beim Starten der verteilten Anwendung ein Fehler auftreten, der die Ausführung unmöglich macht, so wird der aktuelle Teilauftrag für beendet erklärt, indem der P2P-Schicht das Ende der Berechnung mitgeteilt wird und damit signalisiert, dass die Middleware für die nächste Berechnung bereit ist.

4.8.5. Nachladen von Klassen

Es besteht die Möglichkeit, dass eine verteilte Anwendung Zugriff auf selbstgeschriebene Klassen benötigt, die nicht in den normalen Java-Laufzeitumgebungen vorhanden sind. Dafür wird der Anwendungscode als Jar-Archiv verteilt. In diesem Jar-Archiv können weitere Klassen enthalten sein, auf die die verteilte Anwendung frei zugreifen kann.

Probleme treten jedoch dann auf, wenn die verteilte Anwendung über die Objektsendefunktionen der Anwendungsschicht Objekte verschickt, die nur in der Jar-Datei definiert sind. Beim Deserialisieren in der Anwendungsschicht des Empfängers der Nachricht muss im Klassenlader der Middleware diese Klasse sichtbar gemacht werden, da dieser standardmäßig das Jar-Archiv nicht zur Auswertung benutzt. Dafür wurde ein erweiterter Klassenlader definiert, der auch Klassen aus dem Jar-Archiv der verteilten

Anwendung nachladen kann. Das instanziierte Objekt kann dann auf normalem Wege der verteilten Anwendung übergeben werden.

Zum Thema Sicherheit ist bei diesem erweiterten Klassenlader relevant, dass erst versucht wird, eine Klasse über den normalen Klassenlader zu finden und dann im Jar-Archiv gesucht wird, sonst wäre es möglich, der Middleware gefälschte Standard-Klassen „unterzuschieben".

4.8.6. Beenden einer Berechnung

Der verteilten Berechnung stehen mehrere Methoden zur Verfügung, um zu signalisieren, dass die Berechnung abgeschlossen ist. Dabei besteht die Möglichkeit, Ergebnisse der Berechnung an den Initiator übermitteln zu lassen. Die Ergebnisdaten können binär, als Objekte oder auch als String vorliegen. Diese Ergebnisse werden dann im ApplicationHandler falls nötig zu Binärdaten konvertiert und per HTTP-PUT an die URL übermittelt, die in der Beschreibungsdatei angegeben ist. Dazu wird an die Basis-URL noch die Id des Teilpaketes gehängt, zu dem diese Ergebnisse gehören. Bei Aufträgen mit unterschiedlichen Parametern pro Teilpaket ist diese Zugehörigkeit relevant, bei den anderen dient sie nur als Zähler bzw. wird beim Empfänger nicht ausgewertet.

4.8.7. Nachbarschaftslisten

Die Anwendungsschicht verwaltet eine eigene Nachbarschaftsliste, die initial beim Starten des ApplicationHandlers mit den zu dem Zeitpunkt bekannten Knoten gefüllt wird, die an dem gleichen Auftrag rechnen. Es ist wahrscheinlich, dass im Laufe der Zeit mehr Knoten hinzukommen, die an diesem Auftrag rechnen. Damit diese Knoten der Anwendungsschicht auch bekannt gemacht werden können, gibt es die Methode *joinWorkInd()*, mit der die P2P-Schicht diese Aktualisierung der Anwendungsschicht mitteilen kann.

Eine Besonderheit dieser Nachbarschaftsliste ist, dass Knoten hier nicht zufällig wieder herausfallen, wie dies in der implementierten P2P-Schicht der Fall ist. Diese Liste hat keine maximale Länge, Knoten bleiben bis zum Ende der Berechnung in dieser Liste enthalten, außer eine Verbindung zu diesen Knoten schlägt fehl. Ein Knoten, der nicht erreicht werden konnte (zum Beispiel beim Austausch der Nachbarschaftslisten in der P2P-Schicht), wird aus der Liste entfernt.

Da in der Zwischenzeit jedoch Knoten aus der Liste der P2P-Schicht herausfallen können, muss die Anwendungsschicht Sorge dafür tragen, dass ihr die Netzwerkadressen dieser Knoten selber bekannt sind. Die verteilte Anwendung selber kann so weiterhin über die Uids kommunizieren, aber die Anwendungsschicht ist unabhängig vom aktuellen Wissensstand der P2P-Ebene.

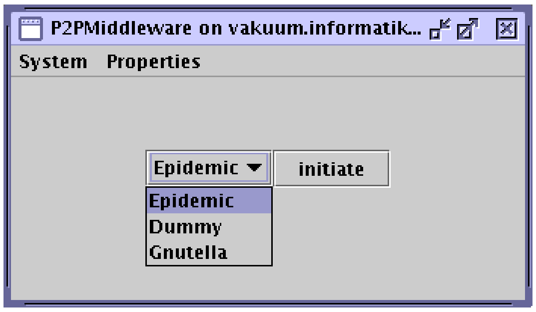

Abbildung 4.8.: Hauptansicht der grafischen Benutzeroberfläche

4.9. P2PGui

Die *P2PGui* ist die Implementierung einer grafischen Benutzerschnittstelle für die Steuerung der Middleware. Mittels dieser Schnittstelle kann ein Nutzer komfortabel einen lokalen Knoten mit einem bestehenden P2P-Netz verbinden, die Verbindung wieder beenden, sowie Aufträge definieren und initiieren. Daneben können die Konfigurationsdateien für die Middleware selber und für das Logging (siehe Abschnitt 4.1.7) grafisch bearbeitet werden.

Die Oberfläche nutzt zur Kommunikation mit der Middleware die *ClientLayer*-Schnittstelle (siehe Abschnitt 3.4.5). Bei der Instanziierung benutzt sie den *DummyClientLayerHandler* als Implementierung des ClientLayer-Interfaces, um die Middleware zu starten. Der DummyClientLayerHandler setzt praktisch nur die Methodenaufrufe durch die P2PGui in entsprechende Aufrufe auf der P2P-Schicht um. Potentiell sind im ClientLayer noch weitere Prüfungen oder Manipulationen der Parameter möglich.

4.9.1. Starten der Middleware

Auf dem Startbildschirm der grafischen Schnittstelle (siehe Abbildung 4.8) muss nur das zu verwendende Protokoll ausgewählt werden, dann kann durch Drücken der Schaltfläche „initiate“ die Middleware mit dem gewählten P2P-Protokoll gestartet werden. Durch Drücken von „connect“ (im Bild durch die Protokollauswahl verdeckt) wird dann die Verbindung zum P2P-Netz hergestellt.

4.9.2. Starten von Aufträgen

Um einen Auftrag starten zu können, muss seine Ausführungsumgebung, d.h. alle benötigten Klassendateien, in einem Jar-Archiv gepackt sein und das Attribut Main-Class im Manifest der Archivs muss auf die benötigte Start-Klasse gesetzt sein. Desweiteren müssen eventuell benötigte Parameterdateien vorbereitet werden.

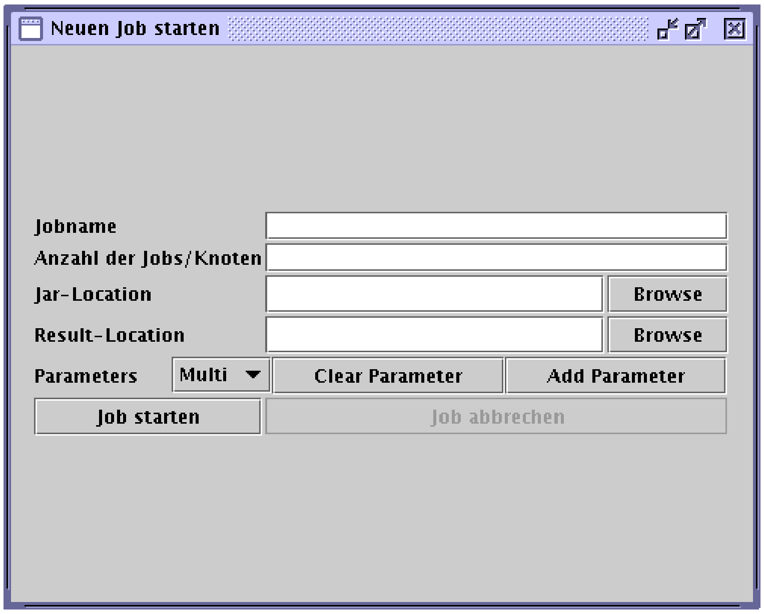

Abbildung 4.9.: Starten eines Jobs mit der grafischen Oberfläche

Das Starten des Auftrags erfolgt dann, indem in der entsprechenden Maske der grafischen Schnittstelle (siehe Abbildung 4.9) ein Name für den Auftrag vergeben wird, die Anzahl der Teilaufträge spezifiziert wird, die Jar-Datei und das Zielverzeichnis für Ergebnisdateien angegeben wird und die Parameterdateien hinzugefügt werden. Dann kann durch Anklicken der Schaltfläche „Job starten" der Auftrag gestartet werden. Zu der Maske für das Starten eines neuen Auftrags kommt man durch Anklicken des Menüeintrages „Neuer Job" im System-Menü. Dieser Eintrag ist jedoch nur dann aktiv und damit nutzbar, solange die Middleware „connected" ist, d.h. solange eine Verbindung zu einem P2P-Netz aufgebaut wurde.

4.9.3. Konfiguration

Durch die grafische Oberfläche können auch die Parameter der Middleware-Implementierung verändert werden. Diese Änderungen wirken sich jedoch nur nach einem Neustart der Middleware auf die Funktionalität aus. Eine Änderung der Logging-Parameter wirkt sich sofort nach dem Speichern aus.

Im Hauptmenü können unter *Properties* Dialoge aufgerufen werden, in denen die Optionen der Middleware bzw. des Loggings geändert werden können. Durch Auswahl des entsprechenden Feldes im Dialog können gewünschte Parameter geändert werden oder durch Anklicken eines Baumeintrages und Drücken der Schaltfläche „Add" auch neue Einträge an dieser Stelle erzeugt werden.

Die Änderung von Parametern der Middleware wird in Abbildung 4.10 gezeigt.

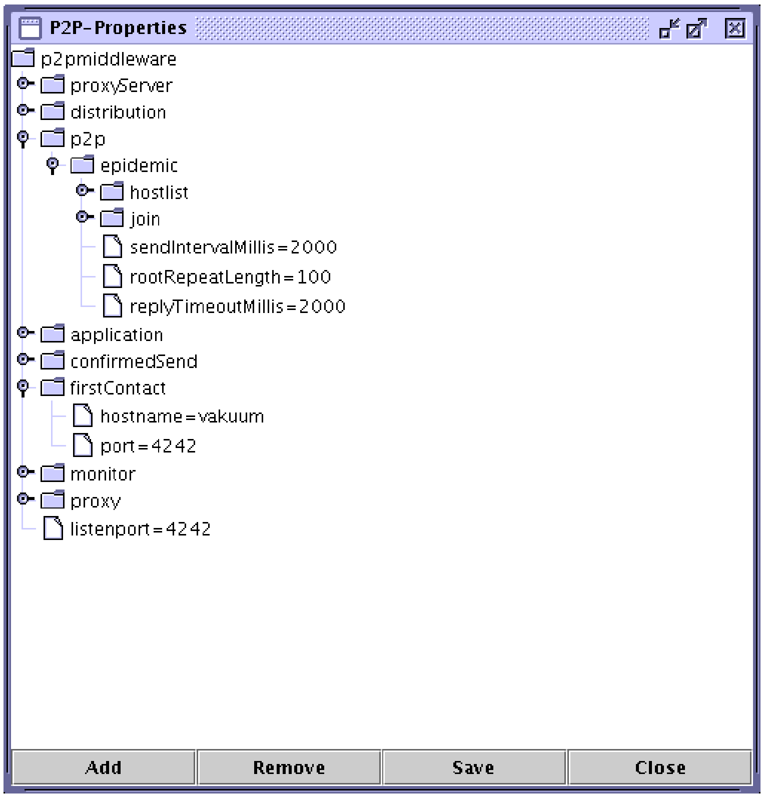

Abbildung 4.10.: Grafisches Ändern von Middleware-Parametern

Eine Übersicht über die verwendeten Parameter findet sich in Anhang A.

5. Ergebnis

In diesem Kapitel soll abschließend ein Überblick über den entwickelten Middleware-Prototypen gegeben werden und dieser mit anderen Systemen verglichen werden. Zudem sollen einige Erweiterungs- und Verbesserungsmöglichkeiten aufgezeigt werden.

5.1. Zusammenfassung

Im Rahmen der vorliegenden Arbeit entstand ein Framework und eine prototypische Implementierung einer Middleware zur Unterstützung von verteilten Anwendungen in P2P-Netzen. Im Rahmen eines epidemischen Protokolls werden in regelmäßigen Abständen Nachbarschaftslisten ausgetauscht, um ein Bild zu erhalten, welche anderen Knoten für Berechnungen zur Verfügung stehen.

Beim Initiieren eines Auftrags sorgt die Middleware dafür, dass alle nötigen Ressourcen auf entfernte Rechner überspielt werden, die Berechnung dort erfolgt und die Ergebnisse zum Nutzer zurück transferiert werden. Die Tatsache, dass die Berechnung verteilt erfolgt, ist für den Dienstnutzer vollständig transparent.

Da der Ansatz P2P-basiert ist, ist jeder Knoten im Netz gleichberechtigt. Jeder einzelne Knoten kann bei Berechnungen mitgenutzt werden und kann selber andere Knoten für Berechnungen mitnutzen. Es gibt für den Nutzer keine Möglichkeit, Knoten für die Berechnung gezielt auszuwählen, da die Knoten indeterministisch mittels eines epidemischen Algorithmus verteilt werden, die Verteilung ist vollständig transparent.

Den Teilprozessen wird eine Kommunikationsschnittstelle geboten, durch welche die Teilnehmer derselben Berechnung miteinander kommunizieren und beispielsweise Zwischenergebnisse der Berechnung austauschen können.

Die entstandene Implementierung umfasst inklusive der vollständigen Dokumentation des Quellcodes mittels Javadoc [SUNa] über 10000 Zeilen Code.

Eigenschaften der Implementierung

Die Implementierung erfüllt die Anforderungen aus den in der Einleitung definierten Zielen der Arbeit (siehe Abschnitt 1.1). Auf die Faktoren Skalierbarkeit und Robustheit soll jedoch noch einmal im Detail eingegangen werden.

Skalierbarkeit

Die entstandene Implementierung ist in gewissem Rahmen skalierbar. Es existiert kein zentraler Dienst, der eine Liste aller aktiven Knoten führt oder versucht, durch Nachrichten solch eine Liste zu ermitteln. Jeder Knoten handelt autonom aufgrund lokaler Informationen. Durch die Verwendung epidemischer Algorithmen bei der Aussendung von Nachrichten wird eine Information im Idealfall innerhalb von $log(n)$ Runden im Netz propagiert.

Einbußen bei der Skalierbarkeit existieren an zwei Stellen:

Auftragsverteilung Neue Aufträge werden ausgehend vom Initiator durch einen initialen Chord-ähnlichen Broadcast (siehe Abschnitt 3.3) und im weiteren Verlauf durch einen epidemischen Algorithmus verteilt. Freie Knoten erhalten die Auftragsbeschreibung und beginnen daraufhin, Anwendungscode und Parameterdateien herunterzuladen. Der Herunterladen des Anwendungscodes geschieht verteilt, d.h. der Code wird nicht immer beim Initiator heruntergeladen, sondern üblicherweise bei dem Rechner, von dem die Auftragsbeschreibung erhalten wurde.

Parameterdateien werden jedoch ausschließlich durch den Initiator verwaltet, um einen genauen Überblick über bereits berechnete Teilaufgaben zu erhalten. Erledigte Aufträge werden somit gar nicht mehr verteilt und die redundante Berechnung möglichst gering gehalten. Wenn jedoch sehr viele Teilaufträge verteilt werden sollen und viele Knoten mitrechnen wollen, kommen auf einen Initiator u.U. sehr viele Anfragen gleichzeitig zu. Eine Überlast des DistributionManagers beim Initiator wird zusätzlich begünstigt, wenn die Parameterdateien für die Teilaufträge sehr groß sind.

Dieser Flaschenhals könnte dadurch behoben werden, dass alle Parameterdateien z.B. durch eine Distributed Hashtable im Netz verteilt gespeichert werden und auch die Ergebnisse an den für die Parameterdateien entsprechenden Knoten gespeichert werden. Über jedes eingegangene Ergebnis könnte der Initiator informiert werden, so dass er dadurch „erfährt“, wann die Berechnung terminiert ist.

Nachrichten zwischen Anwendungen Instanzen derselben verteilten Berechnung können kooperieren, indem sie sich gegenseitig Nachrichten schicken. Für diesen Nachrichtendienst wird ihnen durch die Middleware ein Broadcast- und ein Multicast-Dienst zur Verfügung gestellt. Beide Varianten basieren jedoch darauf, jedem Zielknoten einzeln mittels eines Unicast eine Nachricht zu schicken. Wenn viele Peers an derselben Berechnung beteiligt sind, kann die ausgehende Nachrichtenlast eines Knotens sehr hoch sein. Dieses Problem könnte durch einen effizienten Broadcast- bzw. Multicast-Algorithmus gelöst werden (siehe Abschnitt 2.6 oder Abschnitt 3.3).

Wenn aufgrund der Architektur der verteilten Anwendung jede Instanz häufig Nachrichten an andere Knoten sendet, so ist auch die eingehende Nachrichtenlast erheblich.

Dieses Problem ist jedoch in der Middleware nicht zu lösen, sondern muss in der verteilten Anwendung berücksichtigt werden.

Robustheit

Durch die Verwendung eines P2P-Netzes hat die entstandene Middleware eine gewisse inhärente Form von Robustheit. Jedoch sind Knoten, die eine ausgezeichnete Rolle haben, immer ein Schwachpunkt für die Robustheit eines Systems. So hat auch die implementierte Middleware eine Schwachstelle. Wenn während der Berechnung der Initiator ausfällt, muss dieser seine Berechnung neu starten. Ein kurzzeitiger Netzwerkausfall verursacht kein Problem, solange sich die IP-Adresse des Initiators nicht ändert. Ein längerer Ausfall hat jedoch zwei Effekte:

- Knoten, die die Berechnung starten wollen, können keine Parameterdateien herunterladen und brechen die Berechnung ab.
- Knoten, die ihre Ergebnisse übermitteln wollen, können diese nicht hochladen und verwerfen sie.

Beide Effekte könnten durch den oben angesprochenen Ansatz der Verteilung von Parametern und Ergebnissen in Form einer Distributed Hashtable vermieden werden.

5.2. Vergleich mit bestehenden Arbeiten

An dieser Stelle soll die implementierte Middleware mit einigen der in Abschnitt 2.2 vorgestellten bestehenden Arbeiten verglichen werden und Unterschiede hervorgehoben werden.

5.2.1. SETI@home / BOINC

Wie bereits mehrfach erwähnt ist das Projekt SETI@home bzw. dessen Verallgemeinerung BOINC zwar ein Ansatz für die verteilte Berechnung von Problemen, ist jedoch mit dem hier gewählten Ansatz kaum zu vergleichen. SETI@home ist ein zentralistisches System, bei dem der zentrale Datenserver ein „single point of failure“ ist. Fehler an dieser Stelle beeinflussen das gesamte Netz. Außerdem ist das Spektrum der nutzbaren Anwendungen vergleichsweise gering, da in dieser Umgebung keine Anwendungen berechnet werden können, deren Instanzen miteinander kommunizieren müssen. Ein weiterer Unterschied besteht darin, dass nicht jeder Nutzer, der auch Rechenleistung beisteuert, selber unkompliziert eine Berechnung starten kann.

5.2.2. DREAM

Das DREAM-Projekt liegt in den Anforderungen und in der Realisierung sehr dicht an der vorliegenden Arbeit. Das DREAM-Projekt geht jedoch weiter. Die höheren Anwendungsschichten bei DREAM verwenden evolutionäre Algorithmen und implementieren Schnittstellen, um solche evolutionären Anwendungen mit einfachen Mitteln zu generieren.

Vergleichbar zur vorliegenden Arbeit ist die Basiskomponente DRM. Beide Ansätze nutzen epidemische Algorithmen zur Selbstorganisation des Netzwerks, Knoten agieren alleine aufgrund lokaler Informationen, die sie durch regelmäßigen Austausch mit Nachbarn erhalten. Ebenso ist in beiden Ansätzen die Länge von Nachbarschaftslisten begrenzt, um die Skalierbarkeit des Speicherbedarfs zu gewährleisten.

Unterschiede zu DREAM (der DRM-Teil selber macht keine Annahmen über zu verteilende Anwendungen) ergeben sich in der Art und Weise, wie Aufträge verteilt werden. Beim evolutionären Ansatz von DREAM werden Individuen einer Population zu anderen Populationen geschickt. Knoten agieren als Agenten, die jeweils eigene Inseln betreiben.

5.2.3. OrganicGrid

OrganicGrid basiert ebenso wie diese Arbeit auf P2P-Netzen. Aufträge werden dort jedoch als Agent ins Netz gebracht, teilen sich dort und wandern zu weiteren Knoten. Ein großer Unterschied besteht darin, dass die Agenten den Pfad zum Initiator speichern und die Ergebnisse auf diesem Pfad (darstellbar durch einen Baum) zurückreichen.

Dieser Ansatz ist an diesem Punkt deutlich weniger robust als der in der vorliegenden Arbeit entwickelte Ansatz. Ein Ausfall eines Knoten im entstandenen Baum bewirkt, dass keines der darunterliegenden Ergebnisse zum Initiator zurück wandern kann. Dies bedeutet, dass der Ansatz relativ fehleranfällig ist bzw. Teilaufgaben häufig mehrfach berechnet werden müssen, da Lösungen verloren gehen.

In der entstandenen Implementierung können Ergebnisse auch verloren gehen, jedoch führt der Ausfall eines Knotens immer nur zum Verlust genau eines Ergebnisses. Alle anderen Berechnungen laufen unabhängig davon. Einzig ein Ausfall des Initiators oder ein längerer Ausfall dessen Netzwerkverbindung führt zum Verlust des gesamten Auftrags.

5.2.4. CompuP2P

Der Dienst CompuP2P geht einen Schritt weiter und betrachtet auch Kosten und die Bezahlung von erbrachten Leistungen im P2P-Netz, wobei die Betrachtung nicht nur auf die Bereitstellung von Rechenleistung begrenzt ist, sondern auch die Bereitstellung von Speicherplatz explizit einschließt. CompuP2P erzeugt dynamisch Märkte, auf denen sich freie Ressourcen und Aufgaben treffen und die Allokation eines Auftrags auf einem Knoten ausgehandelt wird. Die Verteilung von Aufträgen geschieht also vollständig unterschiedlich von der vorliegenden Implementierung, da hier die Knoten freiwillig an den Berechnungen teilnehmen.

5.2.5. JXTA

Das Projekt JXTA ist darauf ausgelegt, eine möglichst breite Middleware für verteilte Berechnungen zur Verfügung zu stellen. Prinzipiell wäre JXTA vom Funktionsumfang her eine ideale Plattform auch für diese Arbeit, jedoch ist der geleistete Komfort durch JXTA nur im Tausch gegen Performance zu haben. Bei Tests hat sich herausgestellt, dass der Nachrichtenfluss in JXTA im Vergleich zur implementierten Version deutlich langsamer ist.

Im konkreten Vergleich zum Framework für verteilte Berechnungen von Verbeke et al. [VNRS], das auf JXTA basiert, lässt sich hervorheben, dass dort auch nur unkooperative Berechnungen ausgeführt werden können. Teilnehmende Knoten können nicht miteinander kommunizieren.

Da das Anmelden neuer Knoten im P2P-Netz und das Starten von Aufträgen immer über die gleiche Gruppe in das System erfolgt, dürfte dort jedoch, wie bereits bei der Vorstellung des Frameworks in Abschnitt 2.2 angemerkt, ein Engpass bei hoher Last auftreten, da jedes Mitglied der obersten Monitor-Gruppe diese Nachrichten verarbeiten muss.

5.2.6. G2:P2P

Bei G2:P2P handelt es sich nur begrenzt um ein echtes P2P-System. Potentiell kann zwar jeder Knoten mit jedem Knoten kommunizieren und dort Aufträge starten, jedoch wird die Verteilung der Teilaufträge auf die verschiedenen Rechenknoten alleine durch den Initiator gemacht und über einen .NET-Remoting-Kanal dort gestartet. Dabei muss der Anwender dafür sorgen, dass der auszuführende Anwendungscode auf den Zielrechner gelangt, dieser wird nicht automatisch zum ausführenden Knoten befördert.

5.2.7. Zusammenfassung

Beim Vergleich der entstandenen Implementierung mit anderen bestehenden Projekten zum momentanen Zeitpunkt wird klar, dass es zwar einige Ansätze in der gleichen Richtung gibt, jedoch die Ansprüche und Prioritäten teilweise ganz unterschiedlich sind.

Am vergleichbarsten mit der vorliegenden Middleware dürfte noch DREAM sein, wobei dort das Hauptaugenmerk auf evolutionären Algorithmen liegt und ein Großteil des Projektes sich mit höheren Schichten beschäftigt, um evolutionäre Probleminstanzen zu erzeugen. JXTA arbeitet auf einer ganz anderen Ebene und kann in der Performance nicht mithalten, ist dafür aber auch viel allgemeiner. G2:P2P ist im Funktionsumfang deutlich eingeschränkt und ist fast nicht vergleichbar. Das OrganicGrid ist anfälliger für Knotenausfälle, da dort meist nicht nur eine einzelne Teilberechnung verloren geht, sondern gleich die Ergebnisse des ganzen am ausgefallenen Knoten hängenden Teilbaum nicht zum Initiator zurück kommen. CompuP2P verfolgt durch seine marktzentrische Sicht auch ganz andere Ziele als in der vorliegenden Arbeit erreicht werden sollten.

5.3. Evaluation

Die vorliegende Implementierung wurde mit mehreren Testapplikationen auf Funktionsfähigkeit und Eigenschaften getestet. Konkrete Messungen über die Performance wurden jedoch nicht angestellt. Die Middleware wurde mit drei verschiedenen Anwendungen in einem lokalen Szenario getestet und zwei der Anwendungen in einem weltweiten Szenario.

Die erste Anwendung ist eine reine Testanwendung, die einen ausführenden Knoten 30 Sekunden belegt und dann den IP-Hostnamen des ausführenden Knotens als Ergebnis der Berechnung an den Initiator schickt.

Die zweite Anwendung ist eine verteilte Fraktalberechnung, welche einen bestimmten Ausschnitt eines Fraktals berechnet und diesen berechneten Ausschnitt an den Initiator zurückschickt, welcher das Bild dann zusammensetzt und anzeigt. Bei dieser Anwendung kann jeder Knoten seinen Teilauftrag alleine für sich berechnen, ohne mit anderen Knoten kommunizieren zu müssen. Im Rahmen der verteilen Berechnung eines Fraktals entstand unter anderem Abbildung 5.1.

Die dritte Anwendung ist eine TSP-Heuristik mittels verteilter evolutionärer Algorithmen [FM05]. Im Rahmen der TSP-Berechnung werden Verbesserungen der TSP-Tour durch den ermittelnden Knoten an andere Knoten propagiert. Bei dieser Anwendung machen die Knoten also Gebrauch von den Nachbarschaftslisten und schicken Nachrichten an andere Knoten, die an der gleichen Aufgabe rechnen. Eine errechnete TSP-Lösung ist in Abbildung 5.2 zu sehen. Sie stellt eine Lösung des Problems d15112 dar, welches, wie unschwer erkennbar ist, etwa 15000 Städte in Deutschland enthält.

5.3.1. Lokales Szenario

Das lokale Szenario bestand aus einem PC-Cluster mit 16 Knoten, welche über Gigabit-Ethernet miteinander vernetzt sind. Im Rahmen der Tests wurden die 16 Knoten durch Virtualisierung zu insgesamt 256 Knoten erweitert.

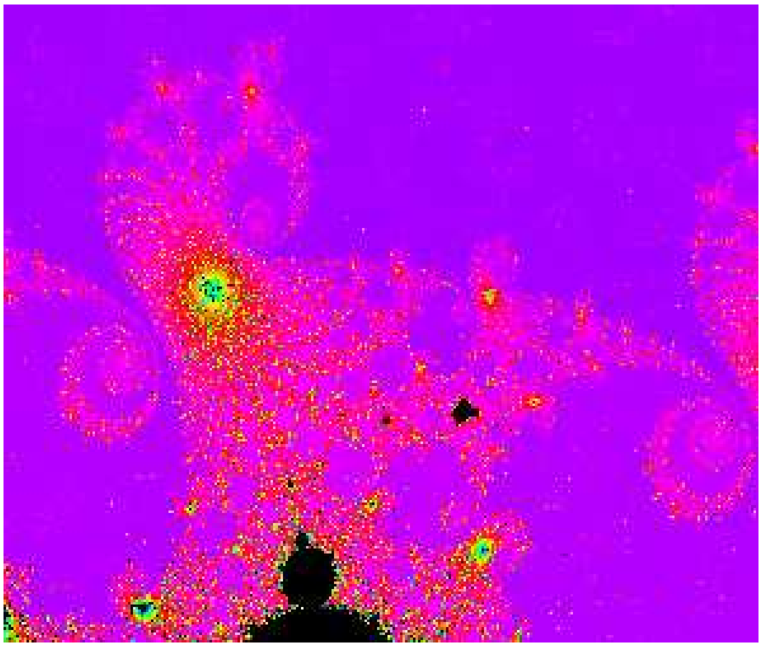

Abbildung 5.1.: Verteilt berechnetes Fraktal

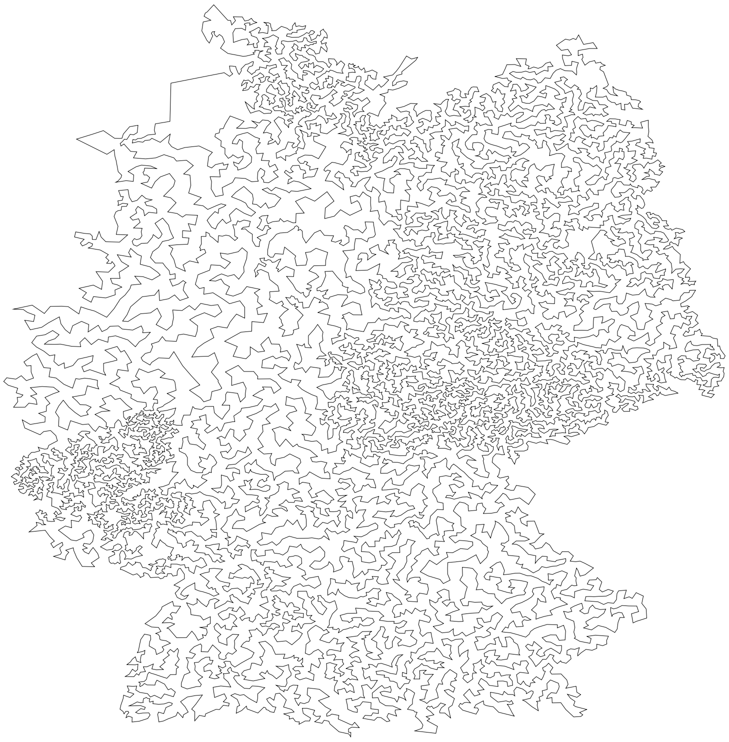

Abbildung 5.2.: Verteilt ermittelte Lösung des TSP-Problems d15112

In diesem Szenario wurden alle drei Anwendungen getestet. In keinem Test mit Nachbarschaftslisten einer Länge größer als 3 Knoten trat eine Separierung des P2P-Netzes auf, d.h. die resultierende Netzwerktopologie war stets ein zusammenhängender Graph. Durch den Test mit der verteilten TSP-Heuristik konnte auch gründlich evaluiert werden, dass eine Kommunikation zwischen Instanzen der verteilten Berechnung erfolgreich funktioniert, auch mit Übertragung von Objekten, die durch die Anwendung definiert und innerhalb der Middleware deserialisiert werden (siehe Abschnitt 4.8.5).

Eine Darstellung der Topologie, die durch einen Überwachungsknoten (siehe Abschnitt 4.7.5) beobachtet wurde, findet sich in Abbildung 5.3 für ein kleines Netz und in Abbildung 5.4 für ein etwas größeres Netz. Die Systeme waren jeweils so konfiguriert, dass die Nachbarschaftslisten maximal die Länge 20 haben konnten. Gerichtete Kanten bezeichnen Nachbarschaftsbeziehungen zwischen Knoten, welche jedoch nicht zwangsweise symmetrisch sein müssen. Die Beschriftung am Beginn einer solchen Kante bezeichnet die Round-Trip-Time (RTT) zu diesem Knoten. Ein Wert von -1 bedeutet, dass die RTT zu diesem Knoten bisher unbekannt ist.

5.3.2. Weltweites Szenario

Das weltweite Szenario wurde durch Einsatz der Middleware im Planet-Lab [CCR+03] ermöglicht. Das Planet-Lab ist ein Zusammenschluss von derzeit etwa 520 Rechnern an 245 weltweit verteilten Standorten. In diesem Kontext wurde die Middleware auf bis zu 60 weltweit ausgesuchten Knoten zum Einsatz gebracht.

Dieses weltweite Szenario wurde mit der einfachen Testapplikation und auch mit der Fraktalberechnung getestet. Beide Anwendungen liefen auch im globalen Maßstab problemlos.

Die Visualisierung einer durch einen Monitorknoten beobachteten Netzwerkstruktur im Rahmen des Planet-Lab sieht man in Abbildung 5.5.

5.3.3. Effizienz

Die Messung der Effizienz der implementierten Middleware ist nicht trivial. Faktoren wie die Grundlast im unbelasteten Fall sind abhängig von Konfigurationsparametern wie Länge der Nachbarschaftsliste und Zykluszeit. Für eine Testumgebung von 17 Knoten und Hostlisten der Maximallänge 20 (d.h. es existierte ein vollvermaschtes Netz) mit einer Zykluszeit von 2 Sekunden betrug die Netzwerklast im Schnitt ca. 14 kbit/s. Auch die Netzwerklast, die durch die Verteilung eines Auftrags induziert wird, ist abhängig von der konkreten Situation im Netzwerk und der Konfiguration. Fest messbar sind Werte wie Kapselungs-Overhead bei Nachrichten zwischen verteilten Anwendungen, haben jedoch nur begrenzte Relevanz und hängen vom Nutzungsprofil der jeweiligen verteilten Anwendung ab.

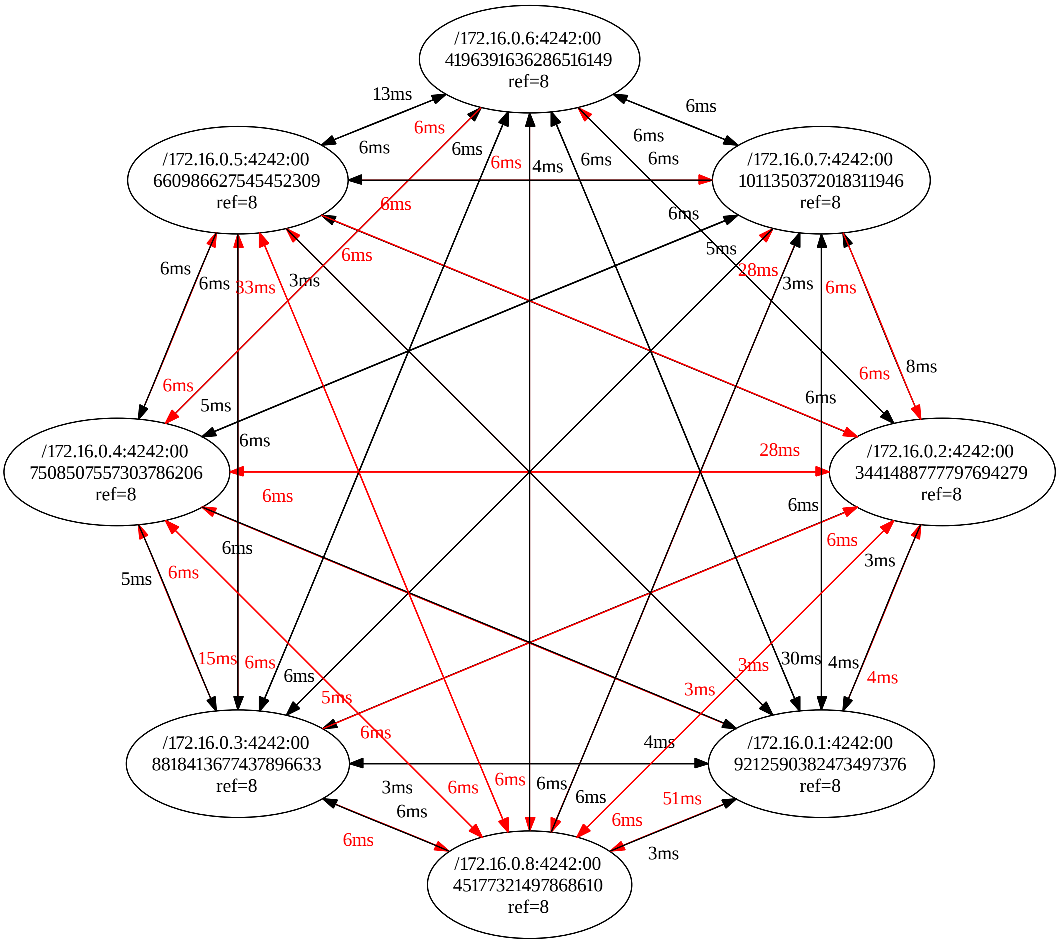

Abbildung 5.3.: Topologie im lokalen Cluster (wenige Knoten)

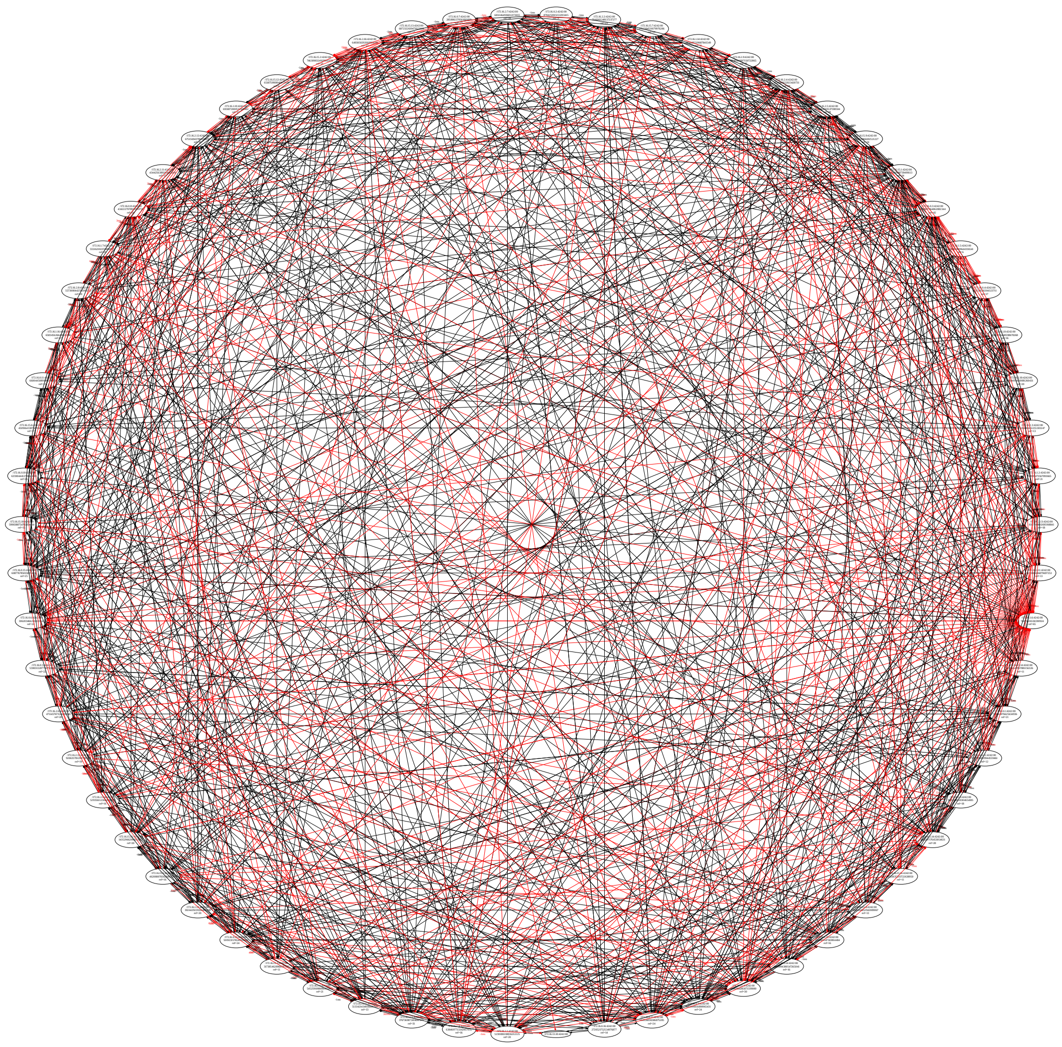

Abbildung 5.4.: Topologie im lokalen Cluster (viele Knoten)

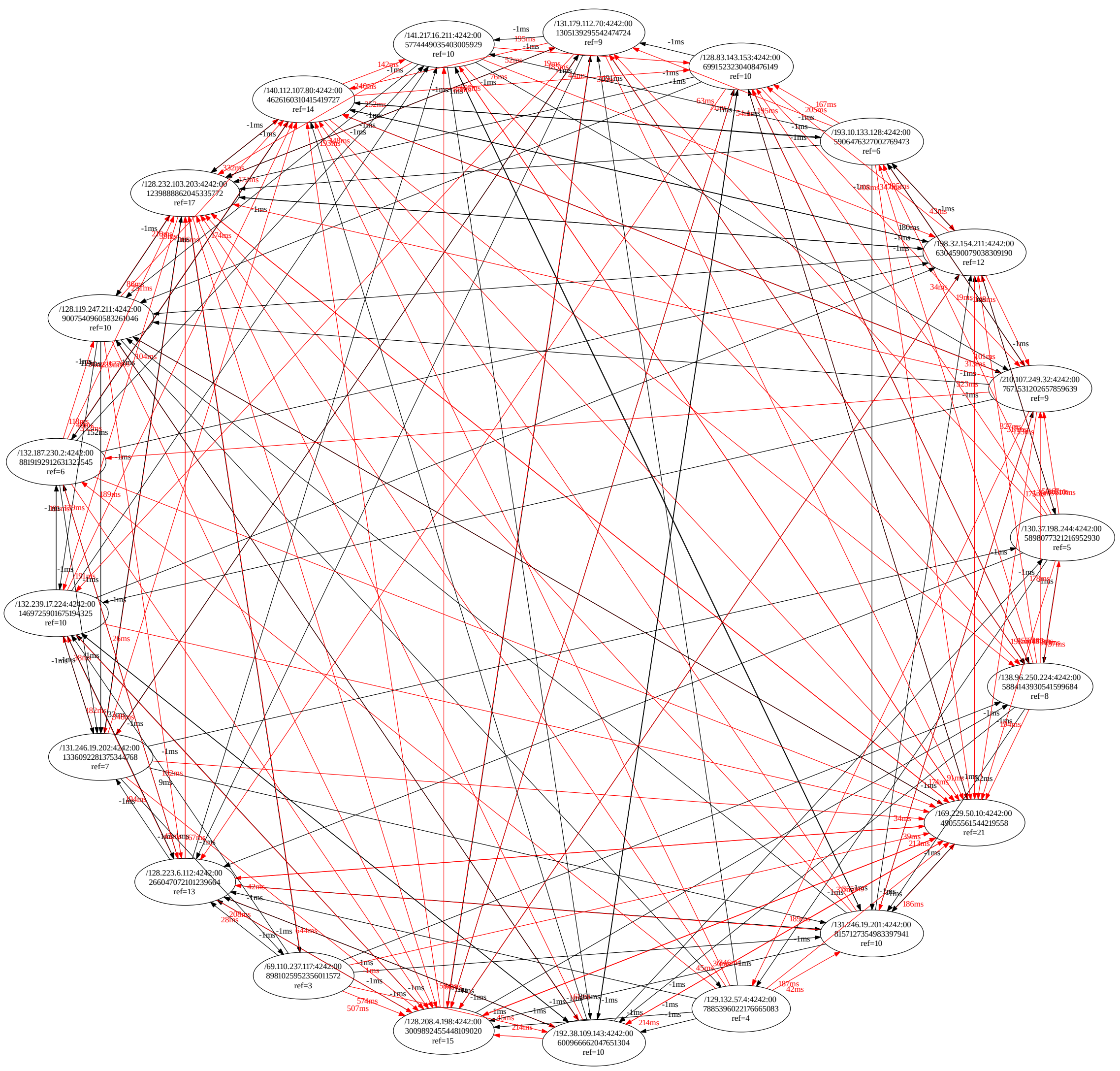

Abbildung 5.5.: Topologie im Planet-Lab

5.4. Erweiterungen und Verbesserungen

In diesem Abschnitt sollen einige Aspekte angesprochen werden, die bei der Implementierung im Rahmen dieser Arbeit zu kurz kamen oder den zeitlichen Rahmen überschritten hätten. Bei breiterem Einsatz der Middleware unter beliebigen Nutzern im Internet müssten vor allem die Sicherheitsaspekte stärker betrachtet werden.

5.4.1. Sicherheitsaspekte

Wie in den Zielen dieser Arbeit bereits angesprochen wird in dieser Arbeit keine Rücksicht auf Sicherheitsfragen genommen. Hauptproblem dabei ist, dass eine verteilte Anwendung derzeit volle Rechte auf den Hostsystemen hat und damit alle Rechte des ausführenden Nutzers genießt. Diese Macht lässt sich in Java jedoch vergleichsweise leicht durch den Einsatz eines *Java Security Managers* einschränken.

Probleme für den Betrieb der Middleware selber existieren aus zwei verschiedenen Richtungen. Zum Einen lässt sich eine Instanz der Middleware durch künstlich außerhalb der Middleware erzeugte Pakete in einen Zustand bringen, in dem auf weitere Daten gewartet wird und die Middleware blockiert, bis diese eintreffen. Solange die Daten nicht kommen, können keine weiteren Pakete verarbeitet werden.

Die andere Gefahr für die Middleware kommt wieder aus der verteilten Anwendung. Derzeit ist es möglich, dass die verteilte Anwendung beliebige Bibliotheken (nicht nur Java-Bibliotheken) nachlädt. Dazu wird das Java-Native-Interface (JNI) genutzt. Sollte jetzt in der nachgeladenen nativen Bibliothek ein Fehler (z.B. eine Speicherverletzung) auftreten, so wird die gesamte Java Virtual Machine durch die Systemumgebung beendet. Auch das Nachladen von Bibliotheken lässt sich leicht durch den Einsatz des Java Security Managers unterbinden. Benötigte Bibliotheken müssen dann nach Java portiert werden, bevor sie benutzt werden können.

5.4.2. Versenden großer Nachrichten

Wie schon im Verlauf der Arbeit erwähnt wurde, ist in der vorliegenden Implementierung die Größe eines Datenpaketes durch die Verwendung einzelner UDP-Pakete auf 64 kB beschränkt. Bei Versenden größerer Nachrichten werden diese bisher verworfen. Die Notwendigkeit für größere Nachrichten besteht z.B. bei der verteilten Berechnung von größeren TSP-Problemen, wenn in der Implementierung des Lösungsalgorithmus ganze TSP-Touren verschickt werden.

Durch die Einführung der Segmentierung von Nachrichten kann dieses Problem gelöst werden. Der Empfänger muss dann warten, bis alle Teilpakete eingetroffen sind und diese in der richtigen Reihenfolge wieder verketten. An dieser Stelle muss damit gerechnet werden, dass ein Paket verloren geht. In diesem Fall kann man einfach das

Gesamtpaket verwerfen. Schwierig ist jedoch die Erkennung, ob eine Nachricht nicht mehr vollständig eintreffen wird. Ein einfacher Ansatz dafür basiert auf der Nutzung von Timern.

5.4.3. Einfache Nutzbarkeit

Trotz einer grafischen Oberfläche ist es im Moment noch einigermaßen aufwändig, die Middleware in Betrieb zu nehmen, ganz abgesehen vom Vorbereiten und Starten von verteilten Berechnungen. Für die Zukunft ist es vorstellbar, einen kleinen schlanken Client in Form eines Applets zu erstellen, der nicht selber am P2P-Netz teilnimmt, sondern der nur in Verbindung mit einem vollwertigen Middleware-Proxy-Server steht und darüber alle Aktionen abwickelt. Dieser Client würde nur als Rechenplattform dienen und keine Möglichkeit haben, eigene Aufträge zu starten. Im Gegenzug dazu müsste er in dem Sinne erweitert werden, dass durch das Applet die aktuelle Berechnung visualisiert wird, damit der Nutzer in irgendeiner ansprechenden Form auch sehen kann, wofür er seinen Rechner gerade zur Verfügung stellt.

Eine weitere Möglichkeit, einen vollwertigen Client zur Verfügung zu stellen, wäre der Einsatz von Java WebStart. Durch den Einsatz von Java WebStart kann der Client einfach durch Anklicken eines Links auf einer Webseite gestartet werden und als vollwertiger Client ohne die Beschränkungen eines Applets agieren.

In beiden Fällen muss zur Schaffung eines Anreizes, Rechenkapazität zur Verfügung zu stellen, die Visualisierung der aktuell ablaufenden verteilten Berechnung verbessert werden. Damit dies möglich wird, müssen die entsprechenden Middlewareschnittstellen um Feedback-Methoden erweitert werden, so dass die Anwendung dem Nutzer Statusinformationen übermitteln kann.

5.4.4. Erweiterungen für Proxy-Clients

Durch die Ziele dieser Arbeit war die Implementierung einer Proxy-Funktionalität vorgegeben, so dass auch Knoten, die sich aus Sicht des P2P-Netzes hinter einer Firewall befinden, an den verteilten Berechnungen teilnehmen können. Diese Forderung ist jedoch nur ein Teilaspekt des Möglichen.

In der vorliegenden Implementierung können sich Knoten über einen Proxy-Server mit dem P2P-Netz verbinden und bekommen eine NetworkId aus dem Adressbereich des Proxy-Servers. Über den aufgebauten TCP-Kanal können Nachrichten empfangen und versandt werden. Es ist jedoch im Moment den vollwertigen Clients vorbehalten, neue Aufträge in das P2P-Netz einspeisen zu können. Bei den Proxy-Clients ist dies nicht realisiert, da ihr lokal gestarteter DistributionManager aufgrund der Firewall nicht erreichbar wäre und die anderen Knoten keine Parameter herunter- und keine Ergebnisse heraufladen könnten.

Die Lösung für dieses Problem liegt darin, dass ein Kommunikationsprotokoll zwischen Proxy-Server und -Client entwickelt wird, über das der Client dem Server alle Auftragsdaten überspielen kann und dieser als Stellvertreter den Auftrag startet. Sobald der Auftrag beendet ist, kann der Proxy-Server den -Client wieder benachrichtigen, dass die Ergebnisse vorliegen und wo sie heruntergeladen werden können.

5.4.5. Skalierbarkeit

Skalierbarkeitseigenschaften und in dem Zusammenhang auftretende Defizite der vorliegenden Implementierung wurden bereits in Abschnitt 5.1 diskutiert.

5.5. Schlussfolgerung

Im Rahmen dieser Arbeit entstand ein System, dass sich durchaus von bisher vorgestellten Arbeiten durch andere Eigenschaften oder eine andere Zielsetzung abgrenzt. Es erfüllt grundlegende Anforderungen an Skalierbarkeit, Verlässlichkeit sowie heterogene Einsatzmöglichkeiten und ist im Hinblick auf Sicherheitsaspekte ausreichend erweiterbar.

Das entstandene Framework, in dessen Rahmen sich der implementierte Prototyp bewegt, ist durch die Definition relativ allgemeiner Interfaces so strukturiert, dass das verwendete P2P-Protokoll ausgetauscht werden kann, die verteilte Anwendung aber nicht umgeschrieben werden muss. Der Initiator kann zur Laufzeit entscheiden, welches P2P-Protokoll in seinem Netz genutzt werden soll. Interoperabel sind verschiedene P2P-Netze jedoch nur, wenn sie dieselben Algorithmen, d.h. dasselbe P2P-Protokoll nutzen.

Die Einsatzfähigkeit der Implementierung wurde anhand von Tests mit verschiedenen verteilten Anwendungen, welche verschiedene Paradigmen nutzen, ausführlich getestet. Sowohl eng als auch lose gekoppelte Anwendungen können effizient verteilt werden.

A. Konfigurationsoptionen

Für die Middleware existieren zahlreiche Konfigurationsoptionen, deren Wirkung hier kurz beschrieben werden sollen. Die verschiedenen Konfigurationsoptionen sind hierarchisch entsprechend ihrer Wirkung aufgebaut. Die einzelnen Teilabschnitte sind hier alphabetisch sortiert.

Der Standardpfad, an dem das System nach einer Konfigurationsdatei sucht, ist die Datei *p2p-properties* im aktuellen Verzeichnis.

A.1. Basiseinstellungen

p2pmiddleware.listenport Hier wird definiert, welchen lokalen UDP-Port die Middleware benutzen darf, um Nachrichten zu empfangen. Dieser Port muss von anderen Knoten als Port der initialen Kontaktadresse angegeben werden, um ein gemeinsames P2P-Netz aufzubauen.

A.2. application

p2pmiddleware.application.max_runtime_seconds (Zeit in Sekunden)
Dieser Parameter definiert, wie viele Sekunden eine verteilte Anwendung laufen darf, bevor sie abgebrochen wird. Relevant ist hier die tatsächliche reale Zeit, nicht die verbrauchten CPU-Sekunden.

A.3. confirmedSend

Die folgenden beiden Parameter steuern das bestätigte Senden. Standardmäßig wird in der Middleware unbestätigt gesendet. Eine verteilte Anwendung kann aber auch einen bestätigten Dienst nutzen.

p2pmiddleware.confirmedSend.maxRetries (Anzahl der Versuche)
Dieser Parameter definiert, wie häufig eine Nachricht wiederholt wird, wenn der Empfänger den Erhalt der Nachricht nicht bestätigt bzw. diese Bestätigung nicht eintrifft.

p2pmiddleware.confirmedSend.waitTimeoutMillis (Zeit in Millisekunden)
Dieser Parameter definiert, wie lange die Middleware auf eine Quittung des Empfängers wartet, bis eine Nachricht erneut gesendet wird (oder der Sendevorgang abgebrochen wird).

A.4. distribution

Die folgenden beiden Parameter beeinflussen den HTTP-Server zur Verteilung des Anwendungscodes und der Parameter sowie zum Empfang von Ergebnissen.

p2pmiddleware.distribution.concurrency (Anzahl von Threads)
Dieser Parameter steuert, wie viele Threads gleichzeitig Anfragen von HTTP-Clients bearbeiten können, d.h. wie viele Verbindungen gleichzeitig offen sind.

p2pmiddleware.distribution.port (Portnummer)
Dieser Parameter steuert, an welchen TCP-Port der HTTP-Server gebunden wird. Ist der Parameter nicht definiert, wird ein zufälliger Port gewählt.

A.5. firstContact

Diese Parameter steuern, welchen Rechner die Middleware initial kontaktiert, um Verbindung zu einem bestehenden P2P-Netz aufzubauen. Möchte der Knoten ein neues P2P-Netz aufbauen, so kann er hier seine eigene Adresse angeben.

p2pmiddleware.firstContact.hostname (Name oder IP-Adresse)
Hier wird der DNS-Name oder die IP-Adresse des Rechners angegeben, der initial kontaktiert werden soll.

p2pmiddleware.firstContact.port (Portnummer)
Hier wird die UDP-Portnummer angegeben, auf dem die Middleware beim initial zu kontaktierenden Rechner läuft.

A.6. monitoring

Hier wird definiert, ob und an welchen Rechner Kontrollnachrichten verschickt werden sollen. Ist einer der beiden Parameter leer bzw. nicht definiert, so werden keine

Kontrollnachrichten verschickt. Die Möglichkeit, Kontrollnachrichten zu verschicken ist nur zu Test- bzw. Debug-Zwecken implementiert und nicht für den normalen Gebrauch vorgesehen. Ein zentraler Überwachungsrechner ist immer ein Flaschenhals und kann mit steigender Zahl von Knoten im P2P-Netz überlasten.

p2pmiddleware.monitor.hostname (Name oder IP-Adresse)
Hier wird der DNS-Name oder die IP-Adresse des Rechners angegeben, an den Kontrollnachrichten geschickt werden sollen.

p2pmiddleware.monitor.port (Portnummer)
Hier wird die Nummer des UDP-Ports angeben, an den der Überwachungsprozess auf dem oben angegebenen Kontrollrechner gebunden ist.

A.7. epidemic

Hier werden die Details der Implementierung des Epidemischen Protokolls definiert.

Allgemeines

In diesem Bereich werden Parameter definiert, die in keine der Unterkategorien passen.

p2pmiddleware.p2p.epidemic.replyTimeoutMillis (Zeit in Millisekunden)
Dieser Parameter definiert, wie lange nach dem Verschicken einer Nachbarschaftsliste auf die Antwort des Empfängers gewartet wird.

p2pmiddleware.p2p.epidemic.rootRepeatLength (Anzahl)
Dieser Parameter legt fest, alle wieviel Runden statt des „normalen" Arbeitspaketes das Wurzelpaket verschickt wird, sofern vorhanden. Das Wurzelpaket ist das Arbeitspaket, das alle Informationen über einen lokal gestarteten Auftrag beinhaltet (vor allem eine Liste der tatsächlich noch fehlenden Teilaufträge). Wird dieser Parameter auf den Wert -1 gesetzt, so wird das Wurzelpaket nie verschickt.

p2pmiddleware.p2p.epidemic.sendIntervalMillis (Zeit in Millisekunden)
Dieser Parameter gibt an, wie häufig der Sendeprozess des epidemischen Protokolls (siehe CyclicSender in 4.7.5) Nachbarschaftslisten und – falls vorhanden – Arbeitsaufträge verschickt. Exakt gibt er die Zeit in Millisekunden an, die der Sendeprozess nach einem erfolgten Sendevorgang (sowohl positiv als auch negativ) wartet, bis der nächste Sendevorgang gestartet wird. Der zeitliche Abstand zwischen zwei Sendevorgängen

beträgt also zwischen $sendIntervalMillis$ und $(sendIntervalMillis + replyTimeoutMillis)$ Millisekunden (zzgl. Verzögerungen durch Multithreading). Diese Zeit sollte höher liegen, als die erwartete Antwortzeit beim Verteilen von Arbeitsaufträgen, damit der Empfänger eines Arbeitspaketes zumindest die Gelegenheit bekommt, einen Auftrag zu bestätigen, bevor dieser dem nächsten geschickt wird.

A.7.1. hostlist

p2pmiddleware.p2p.epidemic.hostlist.size (Anzahl)
Hier wird die geplante Größe der epidemischen Nachbarschaftsliste definiert. Die Liste wird kleiner als hier angegeben, wenn nicht genug Knoten bekannt sind, und sie kann größer werden, wenn der Chord-ähnliche Broadcast (siehe 3.3) dies erfordert.

A.7.2. join

Hier werden Parameter definiert, welche die Join-Prozedur des epidemischen Protokolls beeinflussen.

p2pmiddleware.p2p.epidemic.join.timeout (Zeit in Millisekunden)
Dieser Parameter gibt an, wie viele Millisekunden auf eine Antwort einer JOIN-Nachrichten gewartet wird (positiv oder negativ), bis der Verbindungsversuch entweder wiederholt oder aber abgebrochen wird.

p2pmiddleware.p2p.epidemic.join.tries (Anzahl)
Hier wird definiert, wie häufig versucht wird, eine Verbindung zum P2P-Netz herzustellen, bis der Versuch aufgegeben wird.

A.8. proxy

Dieser Abschnitt konfiguriert, ob der lokale Knoten direkt oder durch einen Proxy mit dem P2P-Netz kommunizieren soll. Wenn einer der beiden Parameter leer gelassen wird, wird kein Proxy-Server benutzt.

p2pmiddleware.proxy.hostname (Name oder IP-Adresse)
Dieser Parameter definiert den DNS-Namen oder die IP-Adresse des Proxy-Server, der für die Netzwerkkommunikation benutzt werden soll.

p2pmiddleware.proxy.port (Portnummer)
Dieser Parameter bestimmt, auf welchem Port der Proxy-Server erreichbar ist.

A.9. proxyServer

Dieser Abschnitt steuert die Bereitstellung eines lokalen Proxy-Servers, um anderen Knoten den Zugang zum P2P-Netz zu ermöglichen.

p2pmiddleware.proxyServer.start (true oder false)
Hier wird konfiguriert, ob eine lokaler Proxy-Server gestartet werden soll.

p2pmiddleware.proxyServer.port (Portnummer)
Hier wird der TCP-Port definiert, auf dem lokale Proxy-Server laufen soll.

A.10. Beispieldatei

Im folgenden sieht man eine Beispiel-Konfigurationsdatei für eine Middleware-Instanz:

```
p2pmiddleware.application.max_runtime_seconds=1800
p2pmiddleware.confirmedSend.maxRetries=20
p2pmiddleware.confirmedSend.waitTimeoutMillis=5000
p2pmiddleware.distribution.concurrency=5
p2pmiddleware.distribution.port=8000
p2pmiddleware.firstContact.hostname=cnode0
p2pmiddleware.firstContact.port=4242
p2pmiddleware.listenport=4242
p2pmiddleware.monitor.hostname=vakuum.informatik.uni-kl.de
p2pmiddleware.monitor.port=14242
p2pmiddleware.p2p.epidemic.hostlist.size=20
p2pmiddleware.p2p.epidemic.join.timeout=1000
p2pmiddleware.p2p.epidemic.join.tries=15
p2pmiddleware.p2p.epidemic.replyTimeoutMillis=2000
p2pmiddleware.p2p.epidemic.rootRepeatLength=100
p2pmiddleware.p2p.epidemic.sendIntervalMillis=2000
p2pmiddleware.proxy.hostname=
p2pmiddleware.proxy.port=
p2pmiddleware.proxyServer.port=8080
p2pmiddleware.proxyServer.start=true
```

B. Fallbeispiel

Hier soll anhand eines einfachen Beispiels ein konkreter Einsatz der Middleware zur verteilten Berechnung von Fraktalen nähergebracht werden.

B.1. Der Client

Der Client dient dazu, die Parameter des Nutzers entgegenzunehmen, den Auftrag zu definieren, in Teilaufträge zu zerlegen und an die Middleware zu übergeben. Er steuert auch Aufbau und Abbau der Verbindung zum P2P-Netz.

Beim Start des Clients wird ein neuer Initial-Task für die Fraktalberechnung angelegt und dieser mit den auf der Kommandozeile angegebenen Parametern gefüttert. Daraufhin wird ein neues Verwaltungsobjekt (*FractalStarter*) erzeugt, dem der neue Task, die Anzahl der zu generierenden Teilaufträge und die Jar-Datei mit der verteilten Anwendung (siehe Abschnitt B.2) übergeben wird. Für alles Weitere ist der Konstruktor dieses Verwaltungsobjektes verantwortlich.

```
public class FractalStarter implements ClientLayerCallback {

    private ClientLayer client;

    private FractalRemoteTask task;

    private File result_dir;

    public static void main(String[] args) {

        if (args.length < 7) {
            System.err.println("zu wenig parameter");
            usage();
            System.exit(1);
        }

        int n = Integer.parseInt(args[0]);
        FractalRemoteTask task = new FractalRemoteTask();
        task.setArgs(args);
```

```
        try {
            FractalStarter starter = new FractalStarter(task, n, args[6]);

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private static void usage() {
        System.out.println("Usage: FractalStarter <job_count> <startRe>"
          + " <startIm> <endRe> <endIm> <depth> <code_file>");
    }
```

Der Konstruktor des Verwaltungsobjektes überprüft initial die Existenz des angegebenen Jar-Archivs und legt für diese Berechnung ein neues temporäres Ergebnis-Verzeichnis an. In diesem Verzeichnis werden eingehende Teilergebnisse gespeichert.

```
public FractalStarter(FractalRemoteTask task, int n, String code_file)
        throws Exception {

    this.task = task;

    File jar = new File(code_file);
    if (!jar.exists()) {
        throw new FileNotFoundException("Jar-File not found");
    }

    this.result_dir = new File(System.getProperty("java.io.tmpdir")
            + "/fractal-" + System.currentTimeMillis());
    this.result_dir.mkdir();
    this.result_dir.deleteOnExit();
```

Im Folgenden wird eine *Map* erzeugt, die Parameter-Indizes (hier *Integer*-Objekte) auf Dateien mit Parametern abbildet. Parameter sind bei dieser Anwendung serialisierte Teil-Tasks. Das Gesamtproblem wird in die gewünschte Anzahl (*n*) von Teilproblemen zerlegt, diese werden einzeln serialisiert, in eine temporäre Datei geschrieben und dann der Parameter-Map hinzugefügt.

```
    Map param_map = new TreeMap();
    File param_file;
    ObjectOutputStream oout;
    Task[] tasks = task.split(n);
    for (int i = 0; i < tasks.length; i++) {
```

```
        param_file = File.createTempFile("fractal-param", "tmp");
        param_file.deleteOnExit();
        oout = new ObjectOutputStream(new BufferedOutputStream(
                new FileOutputStream(param_file)));
        oout.writeObject(tasks[i]);
        oout.flush();
        oout.close();

        param_map.put(new Integer(i), param_file);
    }
```

Zu diesem Zeitpunkt sind alle Vorbereitungen abgeschlossen. Es liegen n Teilprobleme vor, die im P2P-Netz verteilt werden können. Es wird ein neuer Client-Handler gestartet, dem als Parameter mitgeteilt wird, dass die Verteilung mittels eines epidemischen Protokolls erfolgen soll. Zusätzlich wird eine Referenz auf das Verwaltungsobjekt übergeben, damit die Middleware Beginn und Ende der Berechnung mitteilen kann.

Daraufhin wird die Verbindung zum P2P-Netz initiiert. Damit der effiziente Broadcast-Algorithmus genügend Teilnehmer an der Berechnung kennenlernt, wird vor Beginn der Auftragsverteilung 15 Sekunden gewartet. In dieser Zeit erfährt der Knoten von der Existenz weiterer Knoten und kann seine Tabelle für den Chord-ähnlichen Broadcast (siehe Abschnitt 3.3) aufbauen.

Anschließend wird ein neuer Auftrag mit Namen „fractal“ initiiert. Dazu werden als Parameter noch die Anzahl der zu verteilenden Teilaufträge, eine Referenz auf die Jar-Datei mit dem Anwendungscode, eine Referenz auf die oben erzeugte Abbildung von Indizes auf Parameter-Dateien und auf das Ergebnis-Verzeichnis sowie die Art des Auftrags übergeben. Die Art des Auftrags besagt hier, dass jeder Teilauftrag eine eigene Parameterdatei erhält.

```
    this.client = new DummyClientLayerHandler(this, "Epidemic");
    this.client.connect();

    System.out.println("waiting 15s for topology-sync");

    for ( int i=0; i<15; i++) {
        System.out.print(".");
        Thread.sleep(1000);
    }
    System.out.println();

    this.client.newJob("fractal", n, jar, param_map, result_dir,
            WorkSpecFile.MULTI_PARAM);

}
```

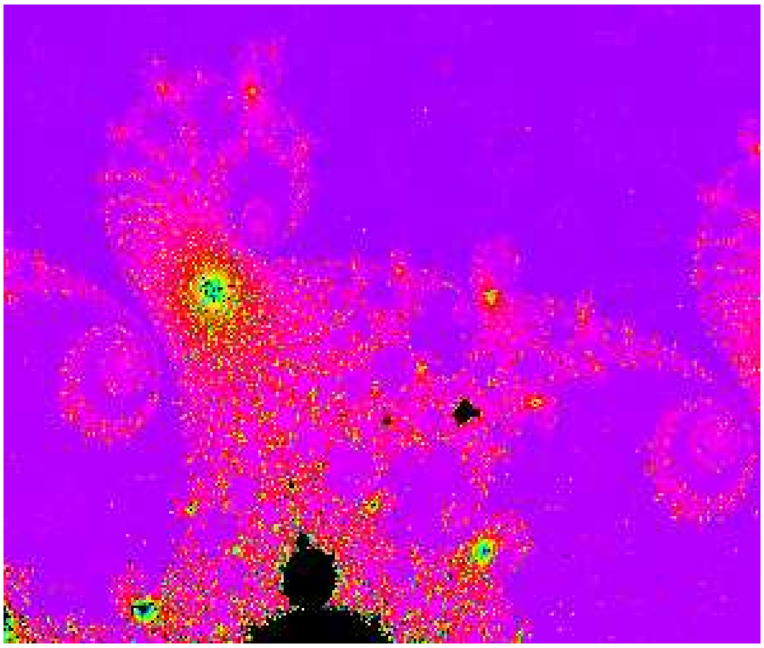

Abbildung B.1.: Verteilt berechnetes Fraktal

Wenn alle Teilaufträge berechnet wurden und die Ergebnisse beim Initiator eingegangen sind, so benachrichtigt die Middleware das Verwaltungsobjekt durch Aufruf der Methode *jobDone()*. Daraufhin können die Teilergebnisse aus dem Ergebnisverzeichnis ausgelesen werden und das Gesamtproblem durch einen Aufruf der Methode *merge()* wieder zusammengefügt werden. Dazu werden die anfangs (vor der Verteilung) serialisierten Teilaufträge wieder aus den Ergebnis-Dateien deserialisiert und instanziiert.

Der so wieder erzeugte vollständig berechnete Gesamtauftrag wird in serialisierter Form zur Visualierung durch ein anderes Programm in einer Datei gespeichert. Das Ergebnis eines Beispielablaufs der Anwendung ist in Abbildung B.1 zu sehen.

Daraufhin wird bei dieser Anwendung die Verbindung zum P2P-Netz wieder abgebaut. Im normalen Einsatz der Middleware sollte die Verbindung nicht abgebrochen werden, damit der Knoten weiterhin für Berechnungen Anderer zur Verfügung steht.

```
public void jobDone(String name) {

    System.out.println("job " + name + " is done");

    File[] res_files = this.result_dir.listFiles();
    FractalRemoteTask[] tasks = new FractalRemoteTask[res_files.length];

    ObjectInputStream oin;
```

```
for (int i = 0; i < tasks.length; i++) {
    System.out.println("reading result from "
    + res_files[i].getName());

    try {
        oin = new ObjectInputStream(new BufferedInputStream(
            new FileInputStream(res_files[i])));
        tasks[i] = (FractalRemoteTask) oin.readObject();

    } catch (IOException e) {
        System.err.println("error reading part of work: "
            + e.getMessage());

    } catch (ClassNotFoundException e) {
        System.err.println(
            "ClassNotFound when reading fractal result: "
            +e.getMessage());
    }
}

this.task.merge(tasks);

File file = new File("fractal-result.bin");

try {

    ObjectOutputStream oout = new ObjectOutputStream(
    new FileOutputStream(file));
    oout.writeObject(this.task);
    oout.close();
    System.out.println("Results saved in file " + file.getPath());

} catch ( IOException e ) {
    System.err.println("io exception saving results: "
        + e.getMessage());
    e.printStackTrace();
}


this.client.disconnect();
this.client.abort();
this.result_dir.delete();
System.exit(0);
```

```
}

public void jobStarted(String name, int jobid) {
    System.out.println("job started");
}
```

B.2. Die Anwendung

Die verteilte Anwendung ist der Code, der im Netz verteilt wird und dort auf den einzelnen Rechenknoten ausgeführt wird. Sie ruft bei der Middleware ihre Parameter ab und deserialisiert daraus den zu bearbeitenden Task. Sollte ein Fehler beim Herunterladen der Parameter aufgetreten sein, so wird der Task abgebrochen.

```
public void start() {

    InputStream param = this.service_layer.getParameters();

    if ( param == null )
        return;
```

Ansonsten berechnet der Task daraufhin seinen Fraktalausschnitt (mittels Aufruf von *runRemote()*) und speichert seine Ergebnisse in lokalen Variablen. Nach Ende der Berechnung wird der Task wieder serialisiert und als Ergebnis an den Initiator geschickt (durch Aufruf der *finished()*-Methode).

```
    try {
        ObjectInputStream oin = new ObjectInputStream(param);
        FractalRemoteTask task = (FractalRemoteTask)oin.readObject();
        task.runRemote();
        this.service_layer.finished(task);
    } catch ( ClassNotFoundException e ) {
        System.err.println("Class not found when unpacking FractalTask: "
          + e.getMessage());
    } catch ( IOException e ) {
        System.err.println("io-exception while receiving FractalTask: "
          + e.getMessage());
    }
}
```

Abkürzungsverzeichnis

CORBA Common Object Request Broker Architecture

DCE Distributed Computing Environment

DCOM Distributed Component Object Model

DNS Domain Name Service

DREAM Distributed Resource Evolutionary Algorithm Machine

HTTP Hypertext Transfer Protocol

I/O Input/Output

IGMP Internet Group Management Protocol

IP Internet Protocol

IRMA Illinois Reliable Multicast Architecture

JNI Java Native Interface

JVM Java Virtual Machine

kB Kilobyte (1024 Byte)

MB Megabyte (1024 kB)

MBONE Multicast Backbone

MPI Message-Passing-Interface

MSC Message Sequence Chart

MTP Multicast Transport Protocol

P2P Peer-to-Peer

PGM Pragmatic General Multicast

PVM Parallel Virtual Machine

RMI Remote Method Invocation

RMTP Reliable Multicast Transport Protocol

RPC Remote Procedure Call

RTT Round-Trip-Time

SETI Search for Extraterrestrial Intelligence

TCP Transmission Control Protocol

TSP Traveling Salesman Problem

UDP User Datagram Protocol

WWW World Wide Web

Literaturverzeichnis

[ACE⁺02] M. G. Arenas, Pierre Collet, A. E. Eiben, Márk Jelasity, J. J. Merelo, Ben Paechter, Mike Preuß, and Marc Schoenauer. A Framework for Distributed Evolutionary Algorithms. In J. J. Merelo Guervós et al., editor, *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature, PPSN VII*, volume 2439 of *Lecture Notes in Computer Science*, pages 665–675. Springer, Berlin, Heidelberg, 2002.

[ACK⁺02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[AFM92] S. Armstrong, A. Freier, and K. Marzullo. RFC 1301: Multicast transport protocol, February 1992. Status: INFORMATIONAL.

[And04] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. *5th IEEE/ACM International Workshop on Grid Computing*, August 2004.

[Atw04] J. Atwood. A classification of reliable multicast protocols. *IEEE Network*, 18(3):24–34, June 2004.

[BE98] Marinho P. Barcellos and Paul D. Ezhilchelvan. An End-to-End Reliable Multicast Protocol Using Polling for Scaleability. In *Proceedings of IEEE INFOCOM'98*, April 1998.

[Ber96] Philip A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[CBL04] Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network. In *Proceedings of the International Conference on Autonomic Computing (ICAC '04)*, pages 96–103, New York, NY, May 2004.

[CCR⁺03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3), July 2003.

[DC85] S. E. Deering and D. R. Cheriton. RFC 966: Host groups: A multicast extension to the Internet Protocol, December 1985. Obsoleted by RFC0988 [Dee86]. Status: UNKNOWN.

[Dee86] S. E. Deering. RFC 988: Host extensions for IP multicasting, July 1986. Obsoleted by RFC1054, RFC1112 [Dee88, Dee89]. Obsoletes RFC0966 [DC85]. Status: UNKNOWN.

[Dee88] S. E. Deering. RFC 1054: Host extensions for IP multicasting, May 1988. Obsoleted by RFC1112 [Dee89]. Obsoletes RFC0988 [Dee86]. Status: UNKNOWN.

[Dee89] S. E. Deering. RFC 1112: Host extensions for IP multicasting, August 1989. Obsoletes RFC0988, RFC1054 [Dee86, Dee88]. See also STD0005 [Pos81e]. Updated by RFC2236 [Fen97]. Status: STANDARD.

[dis] Distributed.Net. `http://www.distributed.net/`.

[EG02] Patrick Thomas Eugster and Rachid Guerraoui. Probabilistic Multicast. In *Proceedings of the 3rd IEEE International Conference on Dependable Systems and Networks (DSN 2002)*, pages 313–322, June 2002.

[EGKM04] P. Th. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie. From Epidemics to Distributed Computing. *IEEE Computer*, 2004.

[Fen97] W. Fenner. RFC 2236: Internet Group Management Protocol, version 2, November 1997. Updates RFC1112 [Dee89]. Status: PROPOSED STANDARD.

[FGM+97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. RFC 2068: Hypertext Transfer Protocol — HTTP/1.1, January 1997. Obsoleted by RFC 2616 [FGM+99]. Status: PROPOSED STANDARD.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, L. Masinter, and P. Leach. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, June 1999. Obsoletes RFC2068 [FGM+97]. Status: PROPOSED STANDARD.

[FM05] Thomas Fischer and Peter Merz. Embedding a Chained Lin-Kernighan Algorithm into a Distributed Algorithm. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS05) (to appear)*, April 2005.

[GMSC03] J. Gemmell, T. Montgomery, T. Speakman, and J. Crowcroft. The PGM reliable multicast protocol. *IEEE Network*, 17(1), February 2003.

[Gon01] L. Gong. JXTA: a network programming environment. *IEEE Internet Computing*, 5:88–95, 2001.

[GS04] Rohit Gupta and Arun K. Somani. CompuP2P: An Architecture for Sharing of Computing Resources In Peer-to-Peer Networks With Selfish Nodes. In *Proceedings of the Second Workshop on the Economics of Peer-to-Peer Systems*, Harvard University, June 2004.

[HB03] Aaron Harwood and Ron Balsys. Peer Service Networks - Distributed P2P Middleware, August 2003. `http://www.cs.mu.oz.au/~aharwood/online/HarwoodBalsys-2003a.pdf`.

[HSS02] Oliver Heckmann, Jens Schmitt, and Ralf Steinmetz. Peer-to-Peer Tauschbörsen: Eine Protokollübersicht. Technical Report TR-KOM-2002-06, Multimedia Communications (KOM), Darmstadt University of Technology, 2002.

[Int95] International Organization for Standardization (ISO). Open Distributed Processing Reference Model. International Standard ISO/IEC IS10746, 1995.

[JPP02] Mark Jelasity, Mike Preuß, and Ben Paechter. A Scalable and Robust Framework for Distributed Applications. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC2002)*, pages 1540–1545. IEEE Press, 2002.

[JXT] JXTA Project. JXTA.
`http://www.jxta.org/`.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201. ACM Press, 2000.

[KHTK00] Sneha Kumar Kasera, Gisli Hjalmtysson, Donald F. Towsley, and James F. Kurose. Scalable reliable multicast using multiple multicast channels. *IEEE/ACM Transactions on Networking*, 8(3), June 2000.

[LHB99] K.-W. Lee, S. Ha, and V. Bharghavan. IRMA: A Reliable Multicast Architecture for the Internet. In *Proceedings of IEEE INFOCOM'99*, April 1999.

[LJLA98] Brian Neil Levine and J.J.Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *ACM Multimedia Systems*, 6(5), September 1998.

[LP96] John C. Lin and Sanjoy Paul. RMTP: A Reliable Multicast Transport Protocol. In *Proceedings of IEEE INFOCOM '96*, pages 1414 – 1424, March 1996.

[LSSP02] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2002.

[MG05] Peter Merz and Katja Gorunova. Efficient Broadcast in P2P Grids. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGRID05) (to appear)*, May 2005.

[MK03] Richard Mason and Wayne Kelly. Peer-To-Peer Cycle Sharing via .NET Remoting, 2003. `http://ausweb.scu.edu.au/aw03/papers/mason/paper.html`.

[Mog84a] J. C. Mogul. RFC 919: Broadcasting Internet datagrams, October 1984. See also STD0005 [Pos81e]. Status: STANDARD.

[Mog84b] J. C. Mogul. RFC 922: Broadcasting Internet datagrams in the presence of subnets, October 1984. See also STD0005 [Pos81e]. Status: STANDARD.

[Mon04] Alberto Montresor. A Robust Protocol for Building Superpeer Overlay Topologies. In *Proceedings of the 4th International Conference on Peer-to-Peer Computing*, Zurich, Switzerland, August 2004. IEEE.

[MP85] J. C. Mogul and J. Postel. RFC 950: Internet Standard Subnetting Procedure, August 1985. Updates RFC0792 [Pos81c]. See also STD0005 [Pos81e]. Status: STANDARD.

[PBS+00] Ben Paechter, Thomas Bäck, Marc Schoenauer, Michele Sebag, A. E. Eiben, Juan J. Merelo, and Terry C. Fogarty. A distributed resource evolutionary algorithm machine (DREAM). In *Proceedings of the Congress on Evolutionary Computation 2000 (CEC2000)*, volume 2, pages 951–958. IEEE, July 2000.

[Pos79] Jon Postel. DOD standard Internet protocol, December 1979.

[Pos80a] J. Postel. RFC 760: DoD standard Internet Protocol, January 1980. Obsoleted by RFC0791, RFC0777 [Pos81b, Pos81a]. Obsoletes IEN123 [Pos79]. Status: UNKNOWN. Not online.

[Pos80b] J. Postel. RFC 768: User datagram protocol, August 1980. Status: STANDARD. See also STD0006 [Pos80c].

[Pos80c] J. Postel. STD 6: User Datagram Protocol, August 1980. See also RFC0768 [Pos80b].

[Pos81a] J. Postel. RFC 777: Internet Control Message Protocol, April 1981. Obsoleted by RFC0792 [Pos81c]. Obsoletes RFC0760 [Pos80a]. Status: UNKNOWN. Not online.

[Pos81b] J. Postel. RFC 791: Internet Protocol, September 1981. Obsoletes RFC0760 [Pos80a]. See also STD0005 [Pos81e]. Status: STANDARD.

[Pos81c] J. Postel. RFC 792: Internet Control Message Protocol, September 1981. Obsoletes RFC0777 [Pos81a]. Updated by RFC0950 [MP85]. See also STD0005 [Pos81e]. Status: STANDARD.

[Pos81d] J. Postel. RFC 793: Transmission control protocol, September 1981. See also STD0007 [Pos81f]. Status: STANDARD.

[Pos81e] J. Postel. STD 5: Internet Protocol: DARPA Internet Program Protocol Specification, September 1981. See also RFC0791, RFC0792, RFC0919, RFC0922, RFC0950, RFC1112 [Pos81b, Pos81c, Mog84a, Mog84b, MP85, Dee89].

[Pos81f] J. Postel. STD 7: Transmission Control Protocol: DARPA Internet Program Protocol Specification, September 1981. See also RFC0793 [Pos81d].

[PSK94] Sanjoy Paul, K.K. Sabnani, and D.M. Kristol. Multicast transport protocols for high speed networks. In *Proceedings of IEEE ICNT'94*, 1994.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[RMB01] William A. Ruh, Francis X. Maginnis, and William J. Brown. *Enterprise Application Integration*. John Wiley & Sons Inc., January 2001.

[RPGE04] Pavlin Radoslavov, Christos Papadopoulos, Ramesh Govindan, and Deborah Estrin. A comparison of application-level and router-assisted hierarchical schemes for reliable multicast. *IEEE/ACM Transactions on Networking*, (3):469–482, June 2004.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[SUNa] SUN Microsystems, Inc. Javadoc Tool Home Page. `http://java.sun.com/j2se/javadoc/reference/docs/index.html`.

[SUNb] SUN Microsystems, Inc. The JAVA™ Programming Language. `http://java.sun.com/`.

[TvS03] Andrew S. Tanenbaum and Martin van Steen. *Verteilte Systeme – Grundlagen und Paradigmen*. Pearson Education Deutschland GmbH, 1 edition, 2003.

[VJvS03] S. Voulgaris, Mark Jelasity, and Martin van Steen. A Robust and Scalable Peer-to-Peer Gossiping Protocol. In *2nd International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2003)*, 2003.

[VNRS] Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, and Ilya Sharapov. Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment. http://www.jxta.org/project/www/docs/mdejxta-paper.pdf.

[wik] Wikipedia. http://www.wikipedia.de/.

[WT00] Brian Whetten and Gursel Taskale. An overview of reliable multicast transport protocol II. *IEEE Network*, February 2000.

[YGM03] Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, March 2003.

[YLY+02] Wonyong Yoon, Dongman Lee, Hee Yong Youn, Seungik Lee, and Seok Joo Koh. A combined group/tree approach for scalable many-to-many reliable multicast. In *Proceedings of IEEE INFOCOM'02*, June 2002.

[ZKJ01] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, University of California at Berkeley, 2001.

Index