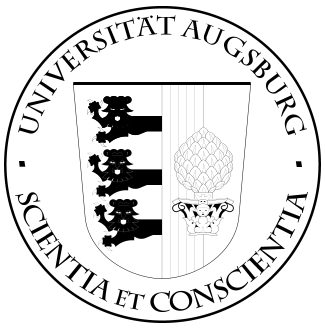
Institut für Informatik
Universität Augsburg



Diplomarbeit

Selbstkonfiguration in einem dienstbasierten Peer-to-Peer Netzwerk

Robert Klaus

Gutachter:	Prof. Dr. Theo Ungerer
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Betreuer:	Wolfgang Trumler
Datum:	6. März 2006

Inhaltsverzeichnis

1 Einleitung

> Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen.
> *Andrew S. Tanenbaum*

Mit diesen Worten formulierte Andrew S. Tanenbaum in seinem Buch [TvS03] die Definition der verteilten Systeme. Diese trifft auf eine Reihe netzwerkbasierter Systeme zu, wie dem Client-Server-Modell und auch neueren Entwicklungen wie Peer-to-Peer Netzwerke. Die Besonderheit dieser Definition ist, dass sowohl eine Problematik als auch ein Ziel von verteilten Systemen darin wieder gespiegelt werden.

Heutige Computersysteme bestehen vielfach aus unterschiedlichsten Systemumgebungen, seien es verschiedene Betriebssysteme, Middleware oder Protokolle, die gekoppelt und mit dem Netzwerk verbunden werden sollen. Dies stellt eine neue Herausforderung an die Administratoren dar, die mit einer ungeahnten Komplexität konfrontiert werden.

Das führt zu der Software Komplexitäts Krise, zuerst von IBM im Oktober 2001 formuliert, dessen Lösung eine ganz neue Art von Herangehensweisen an verteilte Systeme erforderlich macht. Der Lösungsansatz des „Autonomic Computing“ [KC03] orientiert sich am menschlichen Vegetativem Nervensystem. Beispielsweise muss sich das Großhirn nicht um Vorgänge wie den Herzschlag oder die Atmung kümmern, da dies automatisch vom verlängerten Mark gesteuert wird. Ebenso wird das System als eine Art Organismus verstanden, das sich selbstständig um verschiedene Aufgaben kümmert. Das liefert den Schlüssel um der Komplexität Herr zu werden und wird durch vier Prinzipien beschrieben, nämlich Selbstoptimierung, Selbstheilung, Selbstschutz und Selbstkonfiguration.

Diese Prinzipien eignen sich auch für ubiquitäre Systeme, bei denen es sich um viele kleine intelligente Geräte handelt, die den Menschen umgeben und selbstständig oder per Kooperation agieren. Gerade diese hohe Anzahl an verschiedenen Geräten stellt eine besondere Herausforderung dar.

An der Universität Augsburg wird die AMUN Middleware [TPBU05] entwickelt, in der die Prinzipien des Autonomic Computing umgesetzt werden. AMUN soll

die Entwicklung von Applikationen im ubiquitären Umfeld erleichtern. Es findet Einsatz im Smart Doorplates Projekt des Lehrstuhls für Systemnahe Informatik und Kommunikationssysteme. Die Idee ist, dass intelligente Türschilder Besucher durch ein Bürogebäude leiten und sie über digitale Türschilder mit Informationen zu den Personen im Büro versorgen. Mit dieser Arbeit wird ein Prototyp geschaffen, der die notwendigen Grundlagen und Erfahrungswerte zur Implementierung einer Selbstkonfigurations-Strategie liefert.

Die Selbstkonfiguration besteht nach [KC03] aus Richtlinien, die spezifizieren welches Ziel erreicht werden soll, jedoch wird die genaue Ausführung dem System überlassen. Weiterhin wird gefordert, dass sich neue Komponenten nahtlos eingliedern lassen und sich selbstständig gemäß der Gegebenheiten an vorhandener Hard- und Software konfigurieren.

AMUN ist ein dienstbasiertes Peer-to-Peer Netzwerk, in dem zahlreiche Knoten existieren. Eine Applikation, wie sie beispielsweise für das Smart Doorplate Projekt notwendig ist, umfasst eine Reihe von Diensten, die auf passenden Rechnern gestartet werden müssen. Anstatt dies mit einer zentralen Instanz zu erledigen, sollen sich die Knoten selbst konfigurieren und zwar in einer dezentralen Weise, nämlich dass sie selbst aushandeln, welche Dienste von wem ausgeführt werden.

In der folgenden Arbeit werden in Kapitel 2 zuerst ähnliche Arbeiten vorgestellt, um Gemeinsamkeiten und Unterschiede zu zeigen. Die Realisierung und Prüfung des Verfahrens erfolgt außerhalb der Middleware in einem Simulator, dem ein vereinfachtes Modell von AMUN zu Grunde liegt. In Kapitel 3 werden Voraussetzungen für die Modellbildung erarbeitet. Der eigens entwickelte Simulator wird in Kapitel 4 vorgestellt. In Kapitel 5 wird die Nachbildung des sozialen Verhaltens und der daraus resultierende Algorithmus beschrieben. Die verschiedenen Testbedingungen werden in Kapitel 6 erläutert und die Evaluationsergebnisse präsentiert.

2 Ähnliche Arbeiten

Es gibt von Reihe an Arbeiten, die sich mit Sensornetzwerken für ubiquitäre Systeme beschäftigen. Dort werden für Applikation verschiedene Ansätze verfolgt, darunter Middleware [CGG+05] [LM03] [BHG+03] und Mobile Agenten [FRL05], aber ohne die Prinzipien des Autonomic Computing zu verwenden. Diese finden eher Verbreitung in traditionellen Rechnernetzwerken und folgen meist einem zentralisierten Ansatz wie beispielsweise in [Con03].

Im Bereich der Künstlichen Intelligenz gibt es einige Arbeiten zu Kooperation und Teamarbeit (beispielsweise [Tam97] und [PTOK04]), die sich aber mit intelligenten Agenten beschäftigen und sich daher nicht direkt mit dieser Arbeit vergleichen lassen.

Die Verteilung von Diensten anhand verfügbarer Ressourcen ist für diese Arbeit notwendig. Im Bereich der Betriebssysteme wird an der Verwaltung und Zuteilung von Ressourcen zur Maximierung des Gesamtnutzens geforscht, beispielsweise in [RLLS97] und [NM00].

Es existieren nur wenige Arbeiten mit der Thematik der Selbstkonfiguration und meist behandeln diese den Aspekt der Selbstorganisation. Eine Ausnahme findet sich in [BOS01], in dem eine Arbeit zur Selbstkonfiguration des Netzwerks beschrieben wird, die das Problem mit einem zentralen Ansatz löst.

Drei interessante Arbeiten, die sich mit der Selbstkonfiguration befassen, werden im Folgenden vorgestellt.

2.1 Theoretischer Ansatz

Es lohnt sicher immer, eine Lösung mit einem theoretischen Ansatz in Betracht zu ziehen. Meist gibt es bereits Algorithmen und Werkzeuge, die verwendet werden können. Die Selbstkonfiguration enthält eine Anzahl von Bedingungen, die erfüllt werden müssen, weshalb dafür möglicherweise eine Lösung mit *Distributed-Constraint-Satisfaction-Problem* (DCSP) in Frage kommt.

Ein DCSP ist eine Erweiterung des normalen *Constraint-Satisfaction-Problem* (CSP), an dem mehrere Agenten an der Lösung arbeiten, um die Effizienz der Berechnung

zu steigern. Definiert wird es über Variablen, Werte und Bedingungen. Ziel ist es, Belegungen für die Variablen zu finden, so dass alle Bedingungen erfüllt sind.

In der Arbeit *On the Complexity of Distributed Self-Configuration in Wireless Networks* [KWBF00] wird die Problematik der Aufgaben beschrieben, die sich mit einer Selbstkonfiguration in einem multihop Funknetzwerk ergeben, wie es typisch für Sensornetzwerke ist. Diese Aufgaben werden als Distributed-Constraint-Satisfaction-Problems modelliert, welche im Allgemeinen NP hart sind. Es wird bewusst versucht die Probleme mit deterministischen Algorithmen zu lösen, um zu zeigen, dass eine effiziente Berechnung möglich ist. Damit stellt dieser Ansatz das Gegenteil zu dem in AMUN verfolgtem dar.

Es wurden drei praxisrelevante Beispiele gewählt, welche sich mit der Selbstorganisation des Netzwerks beschäftigen und auf theoretische Probleme zurückgeführt werden können. Zu jedem Beispiel wurde eine Modellierung als DCSP gemacht und bestehende Lösungsverfahren darauf angewandt, die mit einer Simulation ausgewertet wurden.

Eines dieser Beispiele ist die Bildung kooperierender Sensorknoten. In einem Sensornetzwerk ist es oftmals erforderlich, dass die Daten mehrerer Sensoren kombiniert werden, um eine sinnvolle Information daraus abzuleiten. Die Bestimmung eines bestimmten Objekts mittels Triangulation wäre ein Beispiel dafür. Dazu müssen mindestens drei Sensoren zusammenarbeiten, welche räumlich verteilt liegen. Über die Sendeleistung der Knoten lassen sich Teilnetze bilden, die eine lokale Einschränkung ermöglichen. Damit befinden sich die kooperierenden Sensoren in Funkreichweite. Es wurde untersucht, wie effizient sich dies mit verschiedenen Bedingungen berechnen ließ.

Dazu gibt es in einem DCSP drei Möglichkeiten: Die Bedingungen können leicht, sehr schwer oder gar nicht erfüllbar sein. Ist ein DCSP leicht erfüllbar oder gar nicht erfüllbar, so kann das relativ schnell berechnet werden. Nur im verbleibenden Fall benötigt die Berechnung eine exponentielle Dauer.

Für die Auswertung wurden die verschiedenen Bedingungen in einem Simulator getestet, wobei weniger als 100 Knoten betrachtet wurden, obwohl die Probleme für sehr große Netze ausgelegt sind. Die vorhandene Rechenkapazität reichte für mehr nicht aus, da im schlimmsten Fall die Kosten für Berechnung und Kommunikation exponentiell mit der Größe des Netzes wachsen.

Würde eine Selbstkonfiguration mit einem DCSP versucht werden, so wäre dies prinzipiell lösbar. Die Verteilung der Dienste ließe sich mit Bedingungen und entsprechenden Variablen darstellen und durch eine verteilte Berechnung auch lösen. Jedoch stellt das eine schwer erfüllbare Bedingung dar, die eine exponentielle Rechenzeit benötigt und ist damit nicht akzeptabel für einen Einsatz in der Praxis.

2.2 Autonomia

Wie der Name schon sagt, ist Autonomia eine Middleware, die auf den Prinzipien des Autonomic Computing basiert[DHX+03]. Es adressiert die Probleme, die in großen, heterogenen Rechnerumgebung entstehen, wie die Anbindung von Komponenten oder neuer Ressourcen. Eine Verwendung für das ubiquitäre Umfeld ist nicht vorgesehen, aber die verwendete Technologie ließe sich prinzipiell auch dafür einsetzen.

Autonomia versucht einige der Eigenschaften des Autonomic Computing umzusetzen, darunter die Selbstkonfiguration. Der Administrator erstellt im Application Management Editor (AME) eine Richtlinie, die von einer Policy Engine verarbeitet wird, die dafür sorgt, dass die notwendigen Aktionen zur Erfüllung der Richtlinie ausgelöst werden. Die eigentlichen Aktionen werden von mobilen Agenten durchgeführt, die von den Autonomic Middleware Services (AMS) bereitgestellt und vom Application Delegated Manager (ADM) überwacht werden.

Application Management Editor
Der Application Management Editor ist ein grafisches Benutzerinterface um eine Applikation mittels vordefinierter Aufgaben und Komponenten zusammenzustellen und die spätere Ausführung zu überwachen. Der Benutzer definiert den Ablauf der Applikation als Richtlinie, in der er die Reihenfolge der Aufgaben modelliert und Voraussetzungen für die benötigten Rechner spezifiziert. Dazu gehört neben den benötigten Ressourcen auch installierte Software, die aus dem Application Information and Knowledge (AIK) abgefragt werden können.

Das Ergebnis wird als Application Service Template im XML-Format wiederum im Repository gespeichert. Das Repository ist ein Datenbank-Server, welcher ebenfalls Strategien zur Problemlösung und andere Verwaltungsaufgaben speichert.

Autonomic Middleware Services
Sobald das Template der Applikation mit dem AME definiert ist, wird eine passende Ausführungsumgebung geschaffen, welche die erforderlichen Ressourcen dynamisch zuteilen kann. Autonomia verwendet ein auf Java und Jini basierendes Mobiles Agenten System das Unabhängigkeit vom Betriebssystem und der Systemarchitektur bietet. Die üblichen Funktionen, wie das Starten von Agenten, die Überwachung der Ausführung und der Transfer von Agenten zu anderen Hosts, werden bereitgestellt. Die dazu notwendige Zugriffsmethode bietet ein eigenes Agenten Transport-Protokoll, welches als Java RMI Stub implementiert ist. Die Methode wird im Resource Repository publiziert, um Abfragen eines Hosts via Jini Lookup Service zu ermöglichen.

Die Kommunikation im System wird über einen Event-Server abgewickelt, welcher JavaSpaces benutzt und über den sich Agenten für Ereignisse registrieren können. Unklar bleibt allerdings, ob die JavaSpaces als persistenter Speicher implementiert

sind, wie sich das System im Fehlerfall verhält und wie die Skalierungsprobleme gelöst werden, die sich in großen Netzwerken ergeben, wie in [TvS03, 798-806] gezeigt wurde.

Application Delegated Manager
Der Application Delegated Manager verwaltet die Applikation zur Laufzeit und ist damit zuständig für die Verteilung der Aufgaben bis hin zur Überwachung der einzelnen Agenten. Der ADM ist also eine zentrale Entscheidungsinstanz.

Es werden einige Information zu den vorhandenen Ressourcen der Rechner im Netzwerk benötigt, um die Selbstkonfiguration durchzuführen. Dazu werden zwei Tabellen erstellt, in denen die Ressourcen der Rechner aufgelistet und die Abhängigkeiten der einzelnen Aufgaben der Applikation ermittelt werden. Die dafür erforderlichen Informationen stammen vom AIK als auch von der AMS. Mittels des Application Service Template und der Tabellen können die Abhängigkeiten aufgelöst werden und die Aufgaben lassen sich im Anschluß mit Mobilen Agenten verteilen. Die Berechnung der Verteilung geschieht also an einer zentralen Stelle.

Autonomia zielt klar auf größere Netzwerke, wie etwa Rechenzentren, um den Administrationsaufwand zu verringern. Dabei ist allerdings nicht klar, wie die Größe des Netzwerks die Performance und die Kommunikation beeinflusst. Zudem entsteht trotz des Einsatzes von Mobilen Agenten unbewusst wieder eine Client Server Architektur. Sowohl das AIK als auch der ADM bilden jeweils eine zentrale Instanz, die für die restlichen Teilen des Systems für Abfragen verfügbar sein müssen. Weder wurden die Gefahren des Ausfalls, noch die Kosten der Kommunikation dieser Instanzen berücksichtigt.

2.3 Autonomic Networked System

Das Autonomic Networked System [BJH+04] ist eine Architektur für die Verwaltung eines ubiquitären Umfelds, wie es etwa im intelligenten Zuhause zu finden wäre. Um die Verwaltung und Verteilung einer Applikation zu vereinfachen, wurde eine Autonomic Middleware entwickelt. Wie bei ubiquitären Systemen üblich, ist es Geräten möglich, jederzeit das Umfeld zu betreten und zu verlassen. Da das System auch mit unterschiedlichen Diensten und Hardware umgehen soll, wird eine Beschreibung erforderlich, die diese soweit vereinheitlicht, dass die Erkennung, Verwaltung, Ausführung und Überwachung von Diensten einfach möglich ist. Dazu wurde eine Lösung mit Semantic Web entwickelt, mit der Dienste und Ressourcen des Netzwerks beschrieben und verknüpft werden können. Möglich wird dies durch eine spezielle Ontologie für Dienstbeschreibungen, der Ontology Language for Services (OWL-S) [The03], die dabei Verwendung findet.

Die Selbstkonfiguration der Geräte im ubiquitären Umfeld nutzen diese Beschreibung, um sich eigenständig zu koppeln. Als Beispiel dient in diesem Fall ein PDA, welcher eine gesicherte Kommunikation zu einem Informationsdienst benötigt. Tritt der PDA in das Netzwerk ein, so wird nach einem passenden Dienst für die Kommunikation gesucht und erst damit ist eine Verbindung zu dem eigentlichen Dienst möglich.

Um besondere Hardware eines Geräts zu nutzen, müssen diese ebenfalls als Dienst abgebildet werden. Beispielsweise würde die Funktionalität eines Sensors zu einem eigenen Dienst werden. Für jede zusätzliche Aufgabe wird ebenfalls ein weiterer Dienst benötigt. Dies wäre etwa der Fall, wenn zur normalen auch eine verschlüsselte Kommunikation zur Verfügung gestellt wird.

Damit existieren im System eine Vielzahl an Diensten, die sich im System registrieren und auf Anfragen antworten.

Das ANS verwendet eine zuverlässige Kommunikation. Selbst wenn ein Gerät nur über UDP sendet, sorgt die Middleware dafür, dass Bestätigungen für eingegangene Nachrichten verschickt werden. Geräte müssen immer über die Middleware kommunizieren, die dafür ein eigenes, einfaches Protokoll bietet. Im Gegensatz zu AMUN, wo versucht wird die Kommunikation möglichst gering zu halten, werden im ANS viele Nachrichten benötigt.

Soziales Verhalten kann als zusätzliche Bedingung ebenso als Dienst spezifiziert werden, um ein zielgerichtetes Verhalten zu bewirken. Als Beispiel dient hier die Selbstheilung, wo ein Dienst aktiv einen Ersatz für einen ausgefallenen Dienst sucht und erst aufhört, wenn er das Ziel erreicht hat und somit die eigentliche Funktionalität wiederhergestellt ist. Leider ist das Hauptziel der Forschung die Kopplung der Basisfunktionalität und somit ist eben das soziale Verhalten ein noch nicht ausgereiftes Nebenziel.

Das ANS ist als direkte Konkurrenz zu AMUN zu sehen. Beide zielen auf das selbe Einsatzgebiet und setzen die Prinzipien des Autonomic Computing um. Die Idee einer Beschreibung aller Dienste mittels einer Ontologie ist sicher ein guter Ansatz, nur führt das zu einer Konzentration auf die Kopplung von Diensten und einer Vernachlässigung eines kooperativen Verhaltens. Zudem fehlt der Zusammenhang zwischen den Diensten und einer Applikation. Es ist nicht definiert, welche Dienste oder Funktionalitäten benötigt werden oder wie diese mit der Selbstkonfiguration eingerichtet werden.

2.4 Zusammenfassung

Die Lösung einer Selbstkonfiguration mit einem Distributed-Constraint-Satisfaction-Problem bringt den Vorteil einer optimalen Lösung wenn auch mit dem Problem, dass dafür ein exponentieller Aufwand notwendig ist. Im Gegensatz dazu wird in AMUN und dieser Arbeit eine nicht-deterministischer Ansatz verfolgt, der zwar keine optimale Lösung liefert, aber die Kosten der Kommunikation drastisch reduziert.

Eine Selbstkonfiguration mit mobilen Agenten verspricht Autonomia zu erreichen. Neben einigen Unklarheiten des Ansatzes wird trotz allem im Prinzip eine Client-Server Architektur verwendet. Probleme mit einem damit entstehenden Single Point of Failure und der Skalierung des Netzwerkes werden nicht berücksichtigt. AMUN dagegen ist als Peer-to-Peer Netzwerk vollständig dezentral organisiert um keinen Single Point of Failure zu haben und eine gute Skalierung zu bieten.

Das Autonomic Networked System verwendet fortgeschrittene Kopplungstechniken um eine Selbstkonfiguration der Dienste in einem ubiquitären System zu erreichen. Die Middleware verwendet eine zuverlässige Kommunikation und konzentriert sich primär auf die Kopplung der Dienste. Dagegen wird in AMUN versucht Nachrichten zu sparen, in dem eine unzuverlässige Kommunikation verwendet wird. Die Selbstkonfiguration in AMUN ist darauf ausgerichtet, die Dienste einer zusammenhängenden Applikation über eine Aushandlung zu verteilen.

3 Modellbildung

Die Selbstkonfiguration für AMUN soll mit einem kooperativen Algorithmus durchgeführt werden. Im ersten Schritt soll dieser in einer Simulation getestet und danach in AMUN implementiert werden. Als Grundlage für die Simulation wird eine modellhafte Abbildung von AMUN benötigt, die annähernd die gleiche Architektur und Eigenschaften aufweist, allerdings ohne die damit verbundene Komplexität.

3.1 Generelle Probleme verteilter Systeme

AMUN ist ein verteiltes System und ist als solches mit den typischen Problemen konfrontiert. Es gibt kein globales Wissen, sondern jeder Knoten ist auf seine eigenen Informationen angewiesen. Ein Beispiel dafür wäre, dass jeder Knoten eine eigene Uhrzeit hat, die unterschiedlich sein kann. Als direkte Konsequenz daraus wird der Zeitpunkt des Versandes einer Nachricht nicht hinzugefügt, da diese Information nur sinnvoll wäre, wenn die Uhren der Knoten zuvor synchronisiert worden wären.

Jeder Knoten ist auf die Informationen angewiesen, die er selbst gesammelt hat oder ihm mitgeteilt worden sind. Eine der Hauptaufgaben dieser Arbeit ist eine Abstimmung zwischen den Knoten durchzuführen, was normalerweise mit sehr viel Nachrichten verbunden wäre oder durch die Wahl einer zentralen Kontrollinstanz, welche die Abstimmung bestimmt. Es wird bewusst auf eine solche Instanz verzichtet, die ein Single Point of Failure darstellt und versucht, das Problem mit Nachrichten in den Griff zu bekommen.

Da die Kommunikation einen hohen Stellenwert hat, muss versucht werden, den Aufwand dafür zu minimieren und trotzdem noch genug Fehlertoleranz bieten zu können, damit am Ende eine konsistente Lösung steht.

3.2 Aufbau von AMUN

Als erster wird ein Blick auf den Aufbau von AMUN geworfen, um daraus bestimmte Voraussetzungen abzuleiten, welche für das Modell wichtig sind. Ausführlichere

Informationen zu AMUN finden sich in [TPBU05].

AMUN besteht aus vier Hauptbestandteilen: Das Transport Interface, dem EventDispatcher, der Diensteschicht und dem Autonomic Manager. Zudem muss noch beachtet werden, wie die Kommunikation realisiert ist.

3.2.1 Transport Interface

Das Transport Interface entkoppelt den Versand der Nachrichten von der unterliegenden Kommunikationsinfrastruktur. Dafür wird in der momentanen Implementierung JXTA verwendet. AMUN setzt eine ungesicherte, asynchrone Kommunikation voraus, womit bestimmte Eigenschaften, wie etwa der Sicherungsschicht von TCP/IP nicht bekannt sind. Es ist aus diesem Grund wichtig, nur die einfachste Funktionalität des Nachrichtentransports für das Modell anzunehmen.

3.2.2 EventDispatcher

Auf jedem AMUN Knoten existiert ein EventDispatcher, der für das Versenden und Empfangen von Nachrichten des Transport Interfaces zuständig ist. Für bestimmte Nachrichtentypen können sich Dienste als Listener eintragen, um beim Empfang einer entsprechenden Nachricht informiert zu werden. Damit ist ein Dienst in der Lage, Informationen zu erhalten ohne selbst einen passenden Dienst finden zu müssen. Ergänzend dazu existiert auch ein Discovery Mechanismus, der sich aber noch in der Entwicklung befindet. Eine so umfangreiche Funktionalität wird für das Modell nicht benötigt. Aus diesem Grund reicht eine Beschränkung auf den notwendigen Teil des Sendens und Empfangens.

3.2.3 Diensteschicht

In AMUN muss jeder Dienst das `Service` Interface implementieren, aber es lassen sich generell zwei Arten unterscheiden, nämlich knotengebundene und bewegliche Dienste. Letztere können zur Selbstoptimierung auf einen anderen Knoten verschoben werden, falls dazu die Entscheidung im Autonomic Manager getroffen wurde. Da die Selbstkonfiguration die Verteilung der Dienste behandelt, wird diese Funktionalität nicht benötigt.

Manche Dienste müssen auf einem speziellen Knoten laufen. Der Grund dafür ist, dass es Dienste geben kann, welche an eine bestimmte Hardware- oder Softwarekomponente gebunden sind, die auf dem Knoten existiert. Das sind Voraussetzungen, die erfüllt sein müssen, damit ein Dienst lauffähig ist.

3.2.4 Autonomic Manager

Als wichtigste Kontrollinstanz ist der Autonomic Manager der Punkt, an dem alle Informationen zusammenfließen. Anhand dieser Informationen kann in Verhandlung mit Autonomic Managern auf anderen Knoten eine Selbstoptimierung erreicht werden. Dazu müssen Teile von AMUN, wie etwa das Transport Interface, überwacht werden. Dafür gibt es spezielle Überwachungsdienste, die Monitore, welche sowohl notwendig als auch nützlich für eine Applikation sind. Die Monitore müssen im Modell berücksichtigt werden, alle anderen Funktionen des Autonomic Managers werden nicht benötigt.

3.2.5 Kommunikation

In vielen Middleware Systemen basiert die Kommunikation auf dem Aufruf von entfernten Methoden, wie etwa durch Remote Method Invocation (RMI) oder Remote Procedure Call (RPC). AMUN dagegen verwendet eine nachrichtenbasierte Kommunikation. Die Nachrichten spielen also eine wesentliche Rolle.

3.2.5.1 Nachrichten

In AMUN werden zwei Arten verwendet, nämlich Unicast- und Broadcastnachrichten. Jede Nachricht kann zu dem noch einen von drei Typen haben.

- **Event** Anzeige eines Ereignisses
- **Request** Anfrage die eine Antwortnachricht fordert.
- **Response** Antwort auf eine Anfrage.

Die Nachrichten bieten an dieser Stelle noch ein paar Feinheiten, welche allerdings für das Modell keine Rolle spielen und deswegen nicht näher betrachtet werden.

3.2.6 Zusammenfassung

Hauptsächlich ist also eine Abbildung der Kommunikation von AMUN und der Infrastruktur notwendig. Dienste und Monitore müssen in der Konfiguration berücksichtigt werden, ebenso wie auch die Abhängigkeit von Hardware.

3.3 Abbildung von AMUN im Modell

3.3.1 Nachrichten

Nachrichten sind ein wichtiger Bestandteil von AMUN und behalten ihre große Bedeutung auch für das Modell.

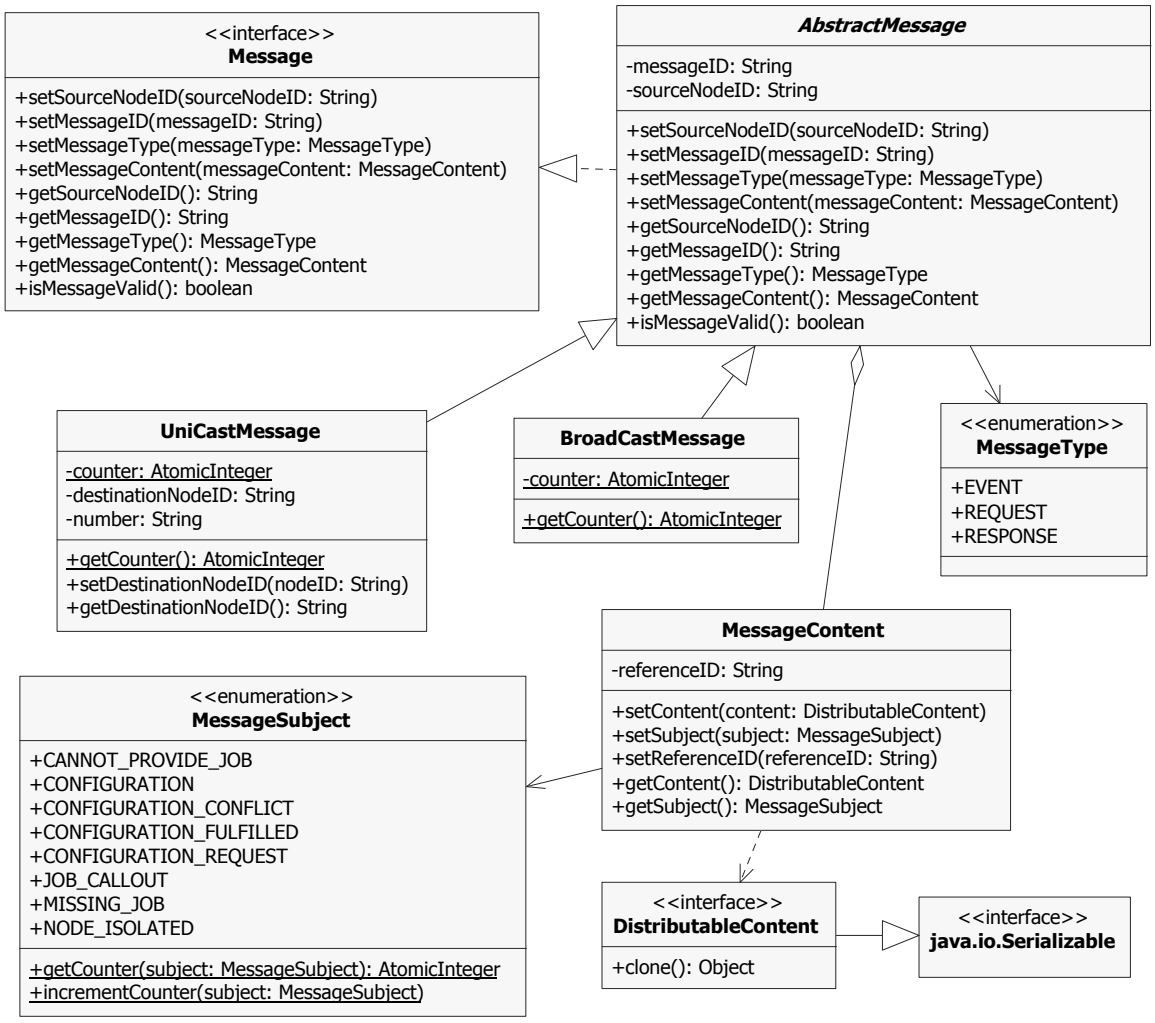


Abbildung 3.1: Aufbau Nachrichten

In Abbildung 3.1 ist der Aufbau der Nachrichten in UML dargestellt. Es existiert eine Schnittstelle `Message`, welche das generelle Verhalten einer Nachricht vorschreibt. Da es sich bei einer Nachricht um einen Datentyp handelt, beschränkt sich dies hauptsächlich auf das Schreiben und Lesen von Werten. Der Vorteil besteht darin, dass sich die Implementierung sehr einfach austauschen lässt. Diese wird zuerst von einer abstrakten Oberklasse und ihren Unterklassen `BroadCastMessage` und

`UniCastMessage` gebildet. Die Unterklassen dienen zur Unterscheidung der Nachrichten und ergänzen für den jeweiligen Fall die notwendigen Daten, so erhält beispielsweise die `UniCastMessage` noch die ID des Empfängers. Damit sowohl Broadcast- als auch Unicastnachrichten gezählt werden können, enthalten beide einen eigenen, statischen Zähler. Dafür wird ein `AtomicInteger` verwendet, der nur durch unteilbare Operationen verändert werden kann. Beim Erzeugen eines Nachrichten-Objekts wird der Zähler inkrementiert. Durch Klonen könnte dieser Wert verfälscht werden, weshalb die notwendige Java Methode in der Oberklasse einen Fehler wirft und nicht überschrieben werden kann.

Jede Nachricht hat einen `MessageType` der dem Nachrichtentyp von AMUN entspricht. Anhand diesem wird im System eine Fallunterscheidung durchgeführt. Um das sicher und einfach handhaben zu können, bietet es sich an, dazu eine Enumeration als Typ zu verwenden.

Das bisher Beschriebene bildet erst einen Rahmen um den `MessageContent`, den eigentlichen Kern einer Nachricht. Zu einer Nachricht gehört ein Betreff und der Inhalt. Da anhand des Betreffs auch Fallunterscheidungen notwendig sind, ist auch dieser ein Enumeration Typ.

Der eigentliche Inhalt ist vom Typ `DistributableContent`, einem Interface, das die Serialisierbarkeit und das Klonen vorschreibt. Das sind sehr wichtige Eigenschaften gerade da die Simulation auf einem Rechner läuft. Es ist überaus wichtig, dass jeder Knoten den Inhalt einer Nachricht als Kopie erhält und nicht die Referenz. Wäre dem nicht so, würde das System gravierende Fehler aufweisen. Beispielsweise wird in der ersten Phase die Konfiguration per Nachricht verteilt. Würde diese nicht geklont werden, so hätte jeder Knoten exakt das gleiche Objekt in den Informationen eingetragen werden. Damit entstünde ein globales Wissen, dass nicht gewünscht ist und auch einen Fehler darstellen würde.

Anders als vielleicht erwartet, wird nicht die gesamte Nachricht geklont, sondern nur der Inhalt. Das geschieht nur bei Bedarf, also wenn auf den Inhalt einer Nachricht zugegriffen wird, liefert dies die Kopie. Der Grund dafür ist, neben dem Schutz des Zählers, sich den Speicher für eventuell ungenutzte Kopien zu sparen.

3.3.2 Kommunikation

Die Kommunikationsinfrastruktur wird durch ein einfaches Netzwerk simuliert und ist in Abbildung 3.2 dargestellt. Jedes Gerät wird dabei als Knoten interpretiert, welches mit einer beliebigen Anzahl an Nachbarknoten verbunden ist. Dieser Zusammenhang wird in der `NetworkGroup` implementiert und wird vom `FakeNetwork` verwendet, um Nachrichten an den beziehungsweise die Adressaten auszuliefern.

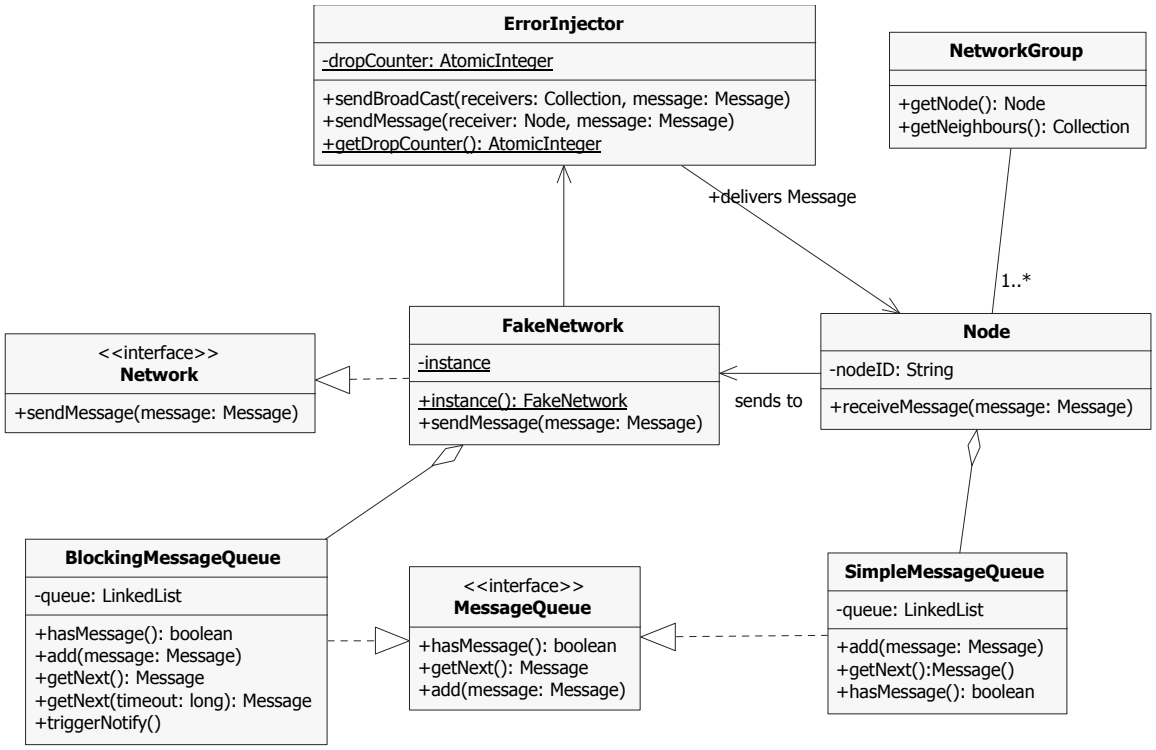


Abbildung 3.2: Kommunikationsinfrastruktur der Simulation

Zum Senden ruft ein Knoten die `sendMessage` Methode des Netzwerks auf, welche die Nachricht in einen Eingangspuffer legt. Das ist notwendig, da jeder Knoten ein Thread ist und der eigentliche Transport nicht im Threadkontext des Senders abgewickelt werden soll. Zudem ist damit auch eine asynchrone Kommunikation gewährleistet.

Das Netzwerk verteilt die Nachricht an die Adressaten, wobei Broadcast Nachrichten entsprechend in einzelne Nachrichten aufgeteilt werden. Da auch das Verhalten des Algorithmus im Fehlerfall prüfbar sein soll, durchläuft jede Nachricht vor der eigentlichen Auslieferung den `ErrorInjector`, der für die Einspeisung von Fehlern verantwortlich ist. Die Nachricht landet schlussendlich im Eingangspuffer des Empfänger-Knotens, der seinen Puffer in jeden Durchlauf prüft und die Informationen verarbeitet. Der Eingangspuffer ist auch hier notwendig aus den gleichen Gründen, die oben genannt wurden.

3.3.3 Fehlereinspeisung

Es gibt verschiedene Fehler, die in der Kommunikation auftreten können. Normalerweise existieren im Schichtmodell des Netzwerkprotokolls der Kommunikationsinfrastruktur passende Sicherungsmechanismen, um diesen Problemen zu begegnen. Wie

bereits erwähnt, wird aber nur eine einfache, ungesicherte Kommunikation vorausgesetzt.

Es kann recht schnell unübersichtlich und kompliziert werden, sollten alle erdenklichen Fehler simuliert werden. Aus diesem Grund ist eine Reduzierung auf wenige und nur notwendige Fehlerarten sinnvoll.

- **Tatsächlicher Verlust**
  Die Nachricht wird nicht ausgeliefert oder kommt beim Adressaten nicht an. Protokolle wie TCP/IP können zwar verlorene Nachrichten erkennen und erneut senden, aber unter der Annahme der einfachen Kommunikation muss davon ausgegangen werden, dass eine Nachricht verloren gehen kann. Damit ähnelt das eher einem Protokoll wie UDP.

- **Verstümmelte Übertragung**
  Der Inhalt der Nachricht wurde bei der Übertragung verändert. Diese können beispielsweise mit Polynomprüfsummen und Paritätsbits erkannt und mit letzteren auch zum Teil behoben werden. Damit ist die Nachricht entweder lesbar oder unbrauchbar, was aber wiederum mit einem einfachen Verlust dargestellt werden kann[1].

- **Verzögerung von Nachrichten**
  Da Latenzzeiten in Netzwerken normalerweise vorhanden sind, sollten diese in der Simulation berücksichtigt werden. Zudem kann es vorkommen, dass die Zeiten durch Routing oder schlechter Anbindung in Teilen des Netzwerks variieren.

- **Überholung von Nachrichten**
  Beim Routen durch das Netzwerk kann es vorkommen, dass eine Nachricht die später losgeschickt wurde auf einer anderen, schnelleren Route zum Ziel geleitet wird und damit schneller ankommt. Das kann durch die oben genannte Verzögerung simuliert werden.

- **Doppeltes Empfangen von Nachrichten**
  Durch einen Fehler kommt die gleiche Nachricht mehrmals an. Das Problem erklärt sich am beliebten Beispiel eines Auftrags zu einer Überweisung, welcher aus Versehen zweimal eingeht und damit das Konto doppelt belastet. Die Lösung ist, dem Auftrag eine eindeutige Nummer zu geben und ebenso wird in AMUN und auch dem Modell verfahren, was dieses Problem eliminiert.

- **Falscher Empfänger**
  Ein Knoten empfängt eine Nachricht, die aber eigentlich an einen anderen

[1] Sofern nicht benötigt wird, dass immerhin eine Nachricht angekommen ist.

Adressaten gehen sollte. Im Regelfall wird diese Nachricht schon vom Protokoll verworfen und in der Simulation kann garantiert werden, dass dieser Fall nicht eintritt.

Die meisten Fehlerarten lassen sich auf zwei reduzieren, dem Verlust einer Nachricht und die Verzögerung einer Nachricht.

Nachrichtenverlust
Es existiert ein Wahrscheinlichkeitswert, mit der jede Nachricht verloren gehen kann. Vor der Auslieferung wird mittels dieses Wertes im `ErrorInjector` der Verlust geprüft. Bei einer Broadcastnachricht wird somit jede Nachricht an die Adressaten dieser Prüfung unterzogen. Zur besseren Kontrolle ist die Verlustrate auf drei Ebenen einstellbar.

- **Gesamtes Netzwerk**
  Jede Nachricht kann mit einer bestimmten Wahrscheinlichkeit verloren gehen.

- **Pro Knoten**
  Simuliert einen unzuverlässigen Knoten, da jede Nachricht an diesen Knoten einer bestimmten Wahrscheinlichkeit verloren gehen kann.

- **Pro Verbindung**
  Simuliert eine gestörte Verbindung

Eine Mischung der Ebenen ist eigentlich nicht vorgesehen, also multiplizieren sich die Fehlerwahrscheinlichkeiten nicht. Falls alle Ebenen mit Werten versorgt werden, dann wird zuerst die Wahrscheinlichkeit des Knotens, dann die der Verbindung und dann die des Netzwerks verwendet.

Nachrichtenverzögerung
Latenzzeiten werden simuliert, in dem eine Nachricht um einen Zeitwert verzögert wird, der zufällig aus einem Intervall gewählt wird, dessen Ober- und Untergrenze eingestellt werden können. Auch hier lassen sich Werte für das gesamte Netzwerk und für einzelne Knoten angeben.

Die Verzögerung wird ebenfalls durch den `ErrorInjector` durchgeführt, in dem die Auslieferung der Nachricht erst zu einem späteren Zeitpunkt erfolgt. Zwar ist auch hier keine Mischung vorgesehen, aber die Latenzzeit eines Knotens wird der des gesamten Netzwerks vorgezogen.

3.4 Konfigurationsbeschreibung

Für die AMUN Middleware werden zwei verschiedene Arten von Diensten verwendet, der `Service` und der `Monitor`. Diese sind Bestandteil einer Applikation für die Middleware und es ist eine Form der Beschreibung notwendig, welche diese spezifiziert und mit der zugleich die Eigenschaft der Selbstkonfiguration erreicht werden soll. Die Beschreibung nutzt das XML-Format und wird für den Simulator in Objekte übersetzt.

3.4.1 Konfigurationsbestandteile

Eine Applikation für AMUN, wie etwa das SmartDoorplate Projekt, verwendet verschiedene Geräte. Die Spanne dieser Geräte reicht von den einfachen Sensorboards über handelsübliche Rechner hin zu richtigen Servern. Manche Funktionalität, wie etwa Sensoren, existieren nur auf bestimmten Geräten. Ihre Informationen sind wichtig und müssen durch entsprechende Dienste ausgewertet werden. Sicherlich ist es zu aufwendig, wenn jedes Gerät beschrieben werden muss. Es wäre in diesem Fall wünschenswert, wenn jedes Gerät selbstständig die Dienste für seine Sensoren startet. Diese Dienste sind gerätgebunden, das heißt, sie sollen auf dem Gerät gestartet werden, dass über die Ressourcen verfügt. Um das zu beschreiben, ist eine Richtlinie notwendig, die genau dies ausdrückt. Dieser Bestandteil der Konfiguration wird *Constraints* genannt.

Neben diesen existieren noch weitere Dienste, welche zwar benötigt werden, aber nicht direkt an ein Gerät gebunden sind. Beispielsweise könnten die gesammelten Informationen durch einen Dienst statistisch ausgewertet werden. Da die meisten Dienste in AMUN verschiebbar sind, wäre es theoretisch möglich, dass ein beliebiger Knoten den Dienst erbringt, aber es wäre sinnvoller, wenn dieser auf einem verhältnismäßig leistungsstarken Rechner laufen würde. In diesem Fall sollte selbstständig der beste Rechner für diesen Dienst ermittelt werden. Dazu werden Systemanforderungen benötigt, welche die notwendigen Voraussetzungen für die Ausführung spezifizieren. Diese Dienste sind der andere Bestandteil der Konfiguration.

Beide Bestandteile sind Grundlage für die Selbstkonfiguration auf die in Kapitel 5 eingegangen wird.

3.4.2 Beschreibung der Constraints

Der Sinn der Constraints lässt sich an einem etwas detaillierteren Beispiel erkennen. In einem Gebäude existiert in jedem Raum ein Rauchmelder und eine Applikation soll

Feueralarm auslösen, wenn dichter Rauch registriert wird. Dazu existiert ein `Monitor` der die Informationen dieser Sensoren ausliest und für die Weiterverarbeitung aufbereitet. Es wäre zwar möglich jeden Knoten einzeln zu beschreiben aber das würde eine Menge Arbeit bedeuten. Viel einfacher wäre es, wenn alle Knoten, die einen Sensor haben den entsprechenden `Monitor` starten.

Würde der letzte Satz formalisiert werden, so würde ein Allquantor verwendet werden um die Vorbedingung zu beschreiben. Ähnliche Konzepte finden sich in der Object Constraint Language (OCL) [Obj05a] einem Teil der Unified Modeling Language (UML) [Obj05b].

Diese Bedingungen heißen in der Konfiguration `constraints` und verwenden eine „forall“ Syntax um das Starten von Diensten abhängig von der Hardware zu ermöglichen.

Ein Beispiel für einen `constraints` Eintrag.

```
<constraints>
  <forall>
    <having>
       <ressource name="IR Sensor" >
          <value name="range" unit="m">2</value>
       </ressource>
    </having>
    <provide>
      <monitor id="656" name="IR Monitoring Service"
        class="de.sik.uau.monitor.IrMonitor" amount="1" />
    </provide>
  </forall>
</constraints>
```

Im obigen Beispiel wird für alle Knoten, welche einen Infrarot Sensor mit einer Reichweite von 2 Metern haben, gefordert einen Monitor zu starten, der diese Ressource überwacht. Im `having` Abschnitt werden Bedingungen definiert, die nicht zwangsläufig auch etwas mit den Diensten im `provide` Teil zu tun haben müssen. In diesem Fall dient das Vorhandensein einer Ressource als Bedingung. Es wäre aber auch möglich einen besonders leistungsfähigen Prozessor vorzuschreiben, um sehr rechenintensive Dienste auf einen bestimmten Knoten auszuführen.

3.4.3 Beschreibung von Diensten

Neben den Constraints gibt es in der Konfiguration einen Abschnitt für Dienste, von denen es in AMUN einmal den `Service` und den `Monitor` gibt. Normalerweise existieren zwischen beiden große Unterschiede. Die Beschreibung beschränkt sich jedoch auf die notwendigen Einträge, um eine Verteilung der Dienste und das Starten zu ermöglichen. In der Konfiguration werden beide getrennt behandelt, da beispielsweise ein `Service` auch einen `Monitor` voraussetzen kann. Beide haben aber einiges gemeinsam und durchlaufen die gleichen Prozesse, weshalb „Job“ als Begriff für beide in dieser Arbeit verwendet wird.

Beispiel einer minimalen Beschreibung:

```
<service id="2" amount="1" name="DataBase"
   class="de.sik.uau.service.SqliteBinding"/>
```

Für die Beschreibung eines Jobs werden zuerst vier Attribute benötigt.

- **ID**
  Ein eindeutiger Wert mit der ein Job identifiziert werden kann.
- **amount**
  Diese Anzahl schreibt vor, von wie viel Knoten der Dienst ausgeführt werden soll.
- **name**
  Eine beschreibende Bezeichnung dieses Jobs für den Administrator.
- **class**
  Für das dynamische Laden des Bytecodes für den `Service` oder `Monitor` wird der Klassenname benötigt.

Diese Beschreibung lässt sich natürlich um Attribute und Elemente erweitern, so können beispielsweise die Systemanforderungen noch hinzugefügt werden, auf die noch später eingegangen wird. Aus der XML-Beschreibung werden später Objekte generiert, welche zur Verwaltung der Dienste dienen. Die Objektstruktur spiegelt diesen Aufbau wieder und wird in Abschnitt 3.4.5 beschrieben.

3.4.4 Abbildung der Systemanforderungen

Sowohl in den Constraints als auch für die Jobs werden manchmal Systemanforderungen verwendet, ähnlich denen, die bei Programmen manchmal angegeben sind.

Diese Anforderungen umfassen Software und Hardware. Da die Abbildung in gleicher Weise erfolgt, beschränkt sich diese Arbeit im Folgenden auf die Hardware.

Die Abbildung der Hardware geschieht durch die Beschreibung der Ressourcen und wird im folgenden Beispiel dargestellt. Jede Ressource (`ressource`) benötigt einen eindeutigen Namen, durch die sie identifizierbar ist und hat mindestens einen oder mehrere Werte (`value`). Der Sinn davon ist, dass auch Hardware abgebildet werden kann, welche durch mehrere Werte beschrieben wird. Beispielsweise könnte es interessant sein, bei Funk neben dem Frequenzband auch die Reichweite zu kennen. Jeder `value` benötigt selbst einen eindeutigen Bezeichner und enthält den Zahlwert und dessen Einheit. Damit wird die Grundlage geschaffen, Ressourcen zu vergleichen und auch damit zu rechnen. Als Gedankenstütze kann die Hardware als eine Menge von Ressourcen und diese als Menge von Werten vorgestellt werden.

Erweitertes Beispiel:

```
<service id="2" amount="1" name="DataBase"
   class="de.sik.uau.service.SqliteBinding">
  <ressource name="RAM">
    <value name="size" unit="MB">256</value>
  </ressource>
  <ressource name="CPU">
    <value name="frequency" unit="Mhz">650</value>
  </ressource>
</service>
```

Jeder Knoten enthält die Beschreibung seiner Hardware, die nach dem gleichen Schema abgebildet ist. Damit kann die Beschreibung der Systemanforderungen nach dem Verteilen der Konfiguration mit der Hardware verglichen werden, um festzustellen, ob und wie gut ein Job ausführbar ist.

Das ist nicht so ohne weiteres möglich. Beispielsweise können ein Pentium IV und ein Athlon 64 Prozessor zwar relativ einfach als „CPU“ klassifiziert werden, wenn es aber darum geht, ihre Leistung anhand der Taktfrequenz zu vergleichen wird es schwierig. Der Athlon ist meist geringer getaktet als der Pentium und somit lassen sich beide nicht direkt mit ihrer Taktfrequenz vergleichen. Aus diesem Grund hat AMD auch die „Quantispeed“ [qua06] Bezeichnung eingeführt, um diesen Vergleich zu ermöglichen.

Werden beide Prozessoren verglichen, so muss irgendwo definiert werden, dass anstatt der realen Taktfrequenz der „Quantispeed“ Wert verwendet werden soll. Mit einer Ontologie, wie beispielsweise der Web Ontologie Language (OWL) [Web04], lässt

sich ein solcher Vergleich realisieren. Für den Einsatz in der Praxis sollte also die Beschreibung der Anforderungen mit einer Ontologie verknüpft werden.

Manchen Anforderungen, wie beispielsweise die benötigte Leistungsfähigkeit des Prozessors, können nicht exakt ermittelt werden, da dies von vielen Faktoren abhängt. In der Praxis werden solche Werte vom Hersteller geliefert oder sie können durch Benchmarks ermittelt werden. Diese können in der Beschreibung verwendet werden.

Trotzdem kann mit dem ermittelten Wert immer noch nicht exakt ausgesagt werden, wie gut der Dienst auf der eingesetzten Hardware läuft. Würde dieser eine ungenügende Leistung erbringen, so schaltet sich die Selbstoptimierung von AMUN ein und könnte beispielsweise den Dienst auf einen anderen Knoten verschieben.

Eine der Aufgaben der Selbstkonfiguration ist es dafür zu sorgen, dass ein Dienst auf einem Knoten gestartet wird, von dem er nicht gleich wieder verlegt werden muss. Damit ist nicht so wichtig, ob die Werte exakt das Laufzeitverhalten wiedergeben, da dies in der Realität nur durch Messungen ermittelt werden kann.

3.4.5 Einsatz der Konfiguration im Modell

AMUN soll auf verschiedenen Hardwareplattformen und auch auf kleineren Rechnern, wie PDAs lauffähig sein. Deswegen sollte die Konfiguration möglichst einfach zu verarbeiten und zu verteilen sein. Weiterhin ist es notwendig, die Anforderungen der Dienste an die Hardware zu beschreiben.

XML bietet sich zur Verwendung an, da es erfolgreich in anderen Systemen als Austauschformat verwendet wird. Es kann sowohl von Maschinen als auch von Menschen gelesen werden und auf vielen Plattformen stehen die notwendigen Tools zur Verarbeitung zur Verfügung. Als weitere Vorteile von XML sind die leichte Erweiterbarkeit und auch die Validierung gegen ein XML-Schema zu nennen.

Die Konfiguration liegt im XML-Format in einer Datei vor und kann von einem Knoten über das Netzwerk geladen werden. Der Inhalt wird von einem Parser verarbeitet und gegen ein XML-Schema verifiziert. Im Anschluß daran, werden die Einträge der Datei in entsprechende Objekte übersetzt.

Damit kann eine Verteilung an alle Knoten im Netzwerk erfolgen. Die Konfiguration wird als serialisiertes Objekt verschickt, obwohl ebenfalls die ursprüngliche Definition in XML-Format dafür dienen kann. Das würde es ermöglichen, umfangreiche Dateien zu komprimieren und somit die Größe der Nachrichten zu verkleinern. Dadurch würde aber, neben einem Parser auf jedem Knoten, auch die Rechenleistung zum Parsen und Validieren anfallen.

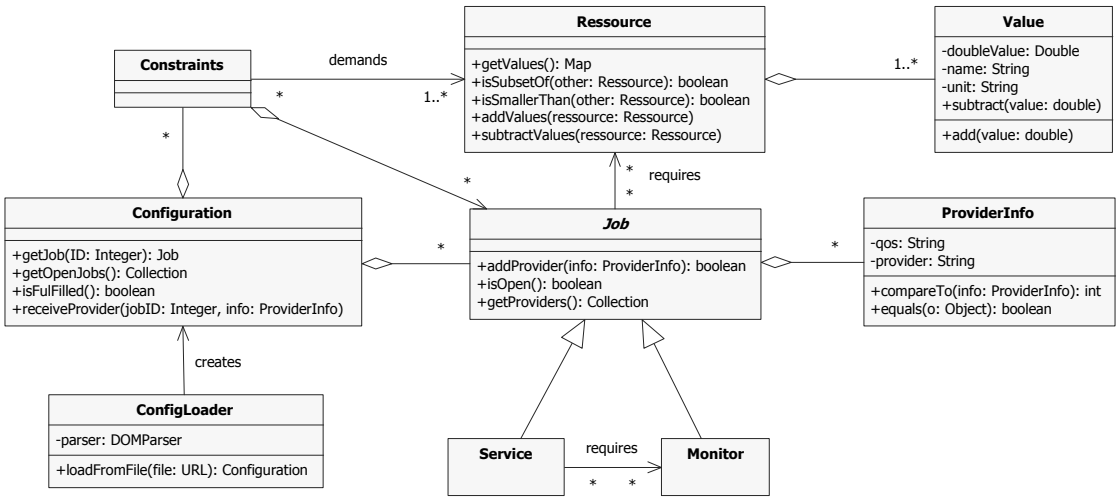


Abbildung 3.3: Klassendiagramm Konfiguration

Abbildung 3.3 zeigt eine UML-Darstellung der Objekte, wie sie vom System zur Laufzeit verwendet werden. Der `ConfigLoader` ist verantwortlich, die XML-Datei zu parsen, zu validieren und die Einträge in entsprechende Objekte umzusetzen. Als Ergebnis wird eine `Configuration` erzeugt. Diese enthält beliebig viele `Constraints` und `Jobs`. `Constraints` enthalten die Anforderungen und die auszuführenden Jobs, wie in Abschnitt 3.4.2 erläutert.

Die `Ressource` ist die Abbildung einer Hard- beziehungsweise Softwareressource und kann sowohl Bestandteil eines `Job` als auch eines `Constraints` sein. Sie kann eine beliebige Anzahl von `Value` enthalten und bietet Methoden um diese zu vergleichen und damit zu rechnen.

Interessant ist die abstrakte Klasse `Job` in Verbindung mit der `Configuration`. Im System dienen beide zur Verwaltung der Informationen der Dienstleister, also der Knoten, die einen Dienst erbringen. Dazu verwendet `Job` eine sortierte Liste, mit der eine beliebige aber feste Anzahl an Dienstleistern verwaltet werden kann. Die Anzahl wird vom Attribut *amount* festgelegt, das bei der Beschreibung des Dienstes angegeben wurde. Informationen zu dem Dienstleister werden in der Klasse `ProviderInfo` festgehalten. Dazu gehören neben der ID des Knotens auch die Quality of Service (QoS), mit der dieser Dienstleister den Dienst erbringen kann.

Ein Job ist offen oder muss noch erbracht werden, wenn die Anzahl der Dienstleister geringer, als in *amount* angegeben ist. Im anderen Fall ist der Job erfüllt. Die `Configuration` ist zuständig für die Verwaltung der `Job` Objekte. Die noch offenen Jobs können abgefragt werden, ebenso ob alle schon erfüllt sind und damit ob die ganze Konfiguration erfüllt ist.

Wird ein Job von einem Knoten erbracht, so wird mit *receiveProvider* die Informatio-

nen in die Liste der Dienstleister eingetragen. Das ist auch dann noch möglich, wenn der Job eigentlich schon erbracht ist. Ansonsten könnten bessere Werte, die später eingegangen sind, nicht mehr eingetragen werden. Die Liste ist sortiert und lässt nur eindeutige Einträge zu. Damit können die schlechten Einträge leicht ermittelt und entfernt werden, was als „Überschreibung“ bezeichnet wird. Es muss noch ermittelt werden, ob der Knoten selbst überschrieben wurde. Ist dies der Fall, so können belegte Ressourcen wieder freigegeben werden.

Die Ressourcen beschreiben die Anforderungen eines Jobs an die Hardware. Der Knoten besitzt aber auch selbst eine Menge an Ressourcen. Um zu prüfen, ob ein Job ausführbar ist, muss die Anforderung eine Teilmenge der vorhandenen Ressourcen und Werte sein. Ist dies erfüllt, so kann mit den Werten gerechnet werden. Wird ein Job ausgeführt, so werden von den vorhandenen Ressourcen die Anforderungen abgezogen, da der Knoten entsprechend belastet wird. Sollen weitere Jobs ausgeführt werden, so werden diese unter Umständen schlechter bewertet. Um dieses ausdrücken zu können, wird eine Metrik benötigt.

3.5 Metrik

Um die Dienstgüte (Quality of Service) eines Dienstes zu bewerten, bedarf es einer Metrik. Diese bildet eine Menge von Eingaben auf einen festen, ganzzahligen Wert ab. Als Eingaben werden die Anforderungen eines Dienstes (Req) und die im System verfügbaren Ressourcen der Hardware (Hw) benötigt. Beide sind eine Menge von Ressourcen (R) und jede Ressource enthält eine Menge von Werten (V).

Um die Ressourcen von Req und Hw vergleichen zu können, wird eine Funktion benötigt, die aussagt, ob sich zwei Werte in Einheit und Art entsprechen, aber nicht den eigentlichen Zahlwert vergleicht. Um zu letzterem eine Aussage zu treffen, werden die gewöhnlichen Vergleichsoperatoren verwendet.

Im Folgenden werden zuerst Beispiele betrachtet, wie die eigentliche Funktion zustande kommt. Dabei wird die Thematik vereinfacht wiedergegeben.

Beispiel 1: Gegeben sei eine Anforderung Req mit einer Ressource CPU die einen Wert V in Höhe 400 vorschreibt. Zwei Knoten (N_1 und N_2) können die Anforderung erfüllen. Die Frage ist, wie gut ist die Quality of Service bei N_1 mit einer entsprechenden Ressource und einem Wert für die CPU von 600 und N_2 mit einem höheren Wert von 2400?

Sicher ist nur, dass N_2, sollte er diesen Dienst erbringen, weitaus weniger belastet wird, als N_1. Somit wäre es wünschenswert, dass bei der Berechnung ein höherer Wert

für N_2 ermittelt wird. In diesem Fall könnten einfach die Differenzen herangezogen werden, also 200 für N_1 und 2000 für N_2.

Damit steht das Ergebnis in direktem Bezug zum Wertebereich der Werte der Ressource. Das führt zu Problemen, wenn mehrere, unterschiedliche Ressourcen verwendet werden.

Beispiel 2: Gegeben sei eine Anforderung *Req* mit zwei Ressourcen. Wie gehabt wird ein Wert für die CPU in Höhe 400 und ein neuer Wert für einen Sensor mit einer Reichweite von 4 verlangt. Diesmal unterscheidet sich N_1 von N_2 bei der Sensor Ressource. N_1 hat zwar immer noch von 600 für die CPU aber einen Wert von 8 für den Sensor. Dagegen hat N_2 einen Wert von 5 für den Sensor.

Damit gibt es zwei Werte, die auf einen abgebildet werden müssen. Wird das durch Aufsummieren der einzelnen Differenzen gelöst, so ergibt sich für N_1 204 und N_2 2001. Obwohl aber N_1 für die Reichweite des Sensors das Doppelte bieten kann, spiegelt sich das in der Quality of Service überhaupt nicht wieder. Da der Wertebereich der CPU um einiges größer ist, verschwindet der Anteil des Sensors in der Berechnung.

Anhand des Quality of Service wird entschieden, welcher Dienst erbracht werden soll. Damit dient die Metrik also auch zum Vergleich von Diensten untereinander. Im obigen Beispiel wäre ein Vergleich schwierig, da der Wertebereich eine bestimmende Rolle spielt. Als Konsequenz muss ein Weg gefunden werden, die Dienste unabhängig vom Wertebereich der Ressourcen zu bewerten.

Das lässt sich einfach lösen. Anstatt der absoluten Werte wird das Verhältnis der Anforderung zu den verfügbaren Ressourcen genommen, um eine Belastung (b) zu berechnen. Die verbleibende Kapazität berechnet sich als $1 - b$. Das lässt sich auch als Quality of Service verwenden, wie man anhand *Beispiel 1* sehen kann. Der Wert von N_1 lässt sich als $1 - \frac{400}{600} = \frac{1}{3}$ und N_2 als $\frac{5}{6}$ berechnen. Damit hat auch N_2 einen höheren Wert und würde den Dienst erbringen.

Diese Werte werden auf eine ganze Zahl abgebildet und mit dem Faktor 100 erweitert. Damit errechnet sich 33 für N_1 und 83 für N_2.

Betrachten wir den interessanteren Fall in *Beispiel 2*, so ergeben sich für N_1 einmal die bereits bekannten $\frac{1}{3}$ und dazu noch $\frac{1}{2}$. Wir bilden in diesem Fall das arithmetische Mittel aus den Werten. Damit ergibt sich letztlich 42 für N_1 und 52 für N_2. Auch in diesem Fall würde N_2 den Dienst erbringen, aber man kann deutlich den Einfluss der zweiten Ressource auf das Ergebnis sehen.

Bisher sieht es so aus, als würde es nur einen positiven Wertebereich für das Ergebnis geben. Die Dienstgüte kann aber auch einen negativen Wert annehmen. Das tritt dann ein, wenn ein Knoten mehr Dienste übernimmt als Ressourcen zur Verfügung stehen. Es ergibt sich die berechtigte Frage, warum das eigentlich möglich ist. Die Metrik

wird verwendet, um einen Job zu bewerten. Ergibt sich ein negativer Wert, so kann der Dienst zwar ausgeführt werden, aber nicht besonders gut. Das hängt wiederum mit den Werten der Ressourcen zusammen. Wird als Beispiel die CPU betrachtet, so kann schlecht angenommen werden, dass eine Aufgabe, welche 600 MHz verlangt, nicht auch von einer CPU mit 500 MHz erbracht werden kann.

Für diesen Fall ist die bisherige Berechnung unzureichend, deswegen gibt es eine Fallunterscheidung die in Formel 3.1 dargestellt ist.

$$b_j = \begin{cases} \frac{V_{j_{req}}}{V_{j_{av}}} & \text{falls } V_{j_{av}} > 0 \\ \frac{V_{j_{max}} + |V_{j_{av}}|}{V_{j_{req}}} & \text{ansonsten} \end{cases} \tag{3.1}$$

Wird das als Gerade interpretiert, so erhält die Funktion durch die Fallunterscheidung im negativen Bereich zugleich eine höhere Steigung. Da die Werte der Funktion stets vergleichbar bleiben, hat das keine negativen Auswirkungen auf den Algorithmus. Im Vergleich zu einem positiven Ergebnis, wird eine Überlastung schlechter bewertet.

Mit der Belastung lässt sich der Quality of Service für eine Ressource mit Formel 3.2 berechnen. Für eine bessere Darstellung kann der errechnete Werte noch mit einem konstanten Faktor c erweitert werden.

$$qos_i = c \cdot \frac{1}{m} \sum_{j=0}^{m} 1 - b_j \tag{3.2}$$

Häufig wird in einer Anforderung mehr als eine Ressource benötigt, weshalb es notwendig ist, einen Wert für alle zu berechnen, was mit Formel 3.3 erreicht wird.

$$qos = \frac{1}{n} \sum_{i=0}^{n} qos_i \tag{3.3}$$

Werden die wichtigsten Eigenschaften betrachtet, so muss für die Metrik gelten, dass sie unabhängig ist vom Wertebereich der Ressourcen. Das bildet die Grundlage für einen Knoten, um zu ermitteln welcher Job gut erbracht werden kann. Obwohl die Hardware jedes Knotens unterschiedlich sein kann, lässt sich damit vergleichen, welcher Knoten einen Job besser erbringen kann.

4 Simulator

Für die Entwicklung und Auswertung wurde ein Simulator geschrieben, mit dem der Ablauf des Algorithmus grafisch mit einer Oberfläche dargestellt werden kann. Es gibt auch die Möglichkeit das Programm vollständig im Batchbetrieb laufen zu lassen was speziell die Auswertung erheblich erleichtert.

4.1 Technische Voraussetzungen

Das Programm benötigt eine Datenbank damit es starten kann. Für diese Arbeit wurde eine MySql Datenbank der Version 4.1 verwendet. Die notwendigen Befehle zur Erzeugung der Tabelle befinden sich in der Datei *create.sql* im Verzeichnis *db*. Die Datenbank ist notwendig, da das System eine Menge an Log Nachrichten produziert, die für eine bessere Handhabbarkeit in der Datenbank gespeichert werden. In Abschnitt 4.3 wird behandelt, welche Einstellungen für die Verbindung und das Logging notwendig sind.

4.2 Oberfläche

Um die grafische Oberfläche zu benutzen, muss das Programm mit dem Kommandozeilenparameter `-simulator` gestartet werden. Es erscheint das Hauptfenster in dem normalerweise die Knoten mit Rechtecken („Label“) dargestellt werden, wie in Abbildung 4.1 zu sehen ist. Startet der Simulator zum ersten Mal wird der Benutzer durch eine Meldung darauf hingewiesen, dass die Definition der Hardware für die Knoten nicht gefunden wurde. Es müssen noch ein paar Einstellungen vorgenommen werden, die für die Ausführung der Simulation wichtig sind. Diese finden sich unter dem Menüpunkt *Optionen*. Dabei handelt es sich um Dateien, die per Dialog ausgewählt werden können.

- **Konfiguration**
  Die Simulation benötigt eine Konfiguration die entweder generiert werden muss

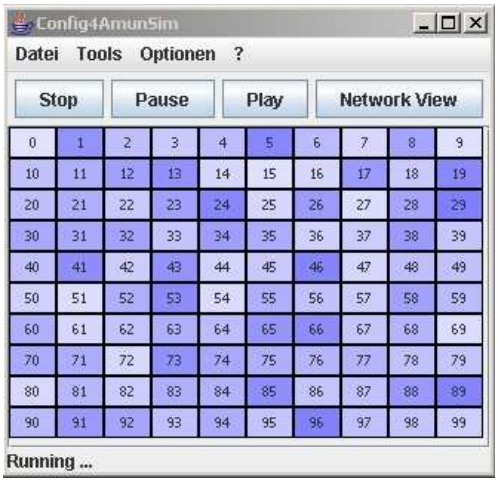


Abbildung 4.1: Hauptfenster des Simulators

oder es wird eine bestehende verwendet. Diese Auswahl wird vor dem eigentlichen Start eines Simulationslaufs gemacht. Mit dieser Option kann die Datei im Vornherein eingestellt werden.

- **Schema**
  Die Datei, die das XML Schema enthält mit dem die Konfiguration validiert wird.

- **Mapping Datei**
  Für den Simulator werden viele Objekte in XML serialisert. Damit die Serialisierung korrekt abläuft sind ein paar Übersetzungsdefinitionen notwendig, die sich in der *Mapping Datei* befinden. Näheres dazu findet sich in Abschnitt 4.5.

- **Default Hardware**
  Die gesamte Hardware, die für die Simulation verfügbar ist, befindet sich in dieser Datei. Damit kann beim Erstellen von Knoten Ressourcen und ihre Werte hinzugefügt und editiert werden. Die Definitionen in der Datei lassen sich leicht editieren, da sie im XML-Format geschrieben sind. Die Datei lässt sich natürlich ebenso durch eine andere ersetzen, was beispielsweise für die Testläufe wichtig ist, die in Kapitel 6 beschrieben sind.

Sind diese Einstellungen vorgenommen, kann ein Simulationslauf gestartet werden. Dazu wird der Unterpunkt *Neu* im Menü *Datei* ausgewählt. Der Simulator generiert das Netzwerk und die Knoten. Dazu werden einige grundlegende Werte benötigt, die per Dialog eingestellt werden, wie in Abbildung 4.2 zu sehen.

- **Anzahl Knoten**
  Die Anzahl der Knoten, die am Simulationslauf teilnehmen.

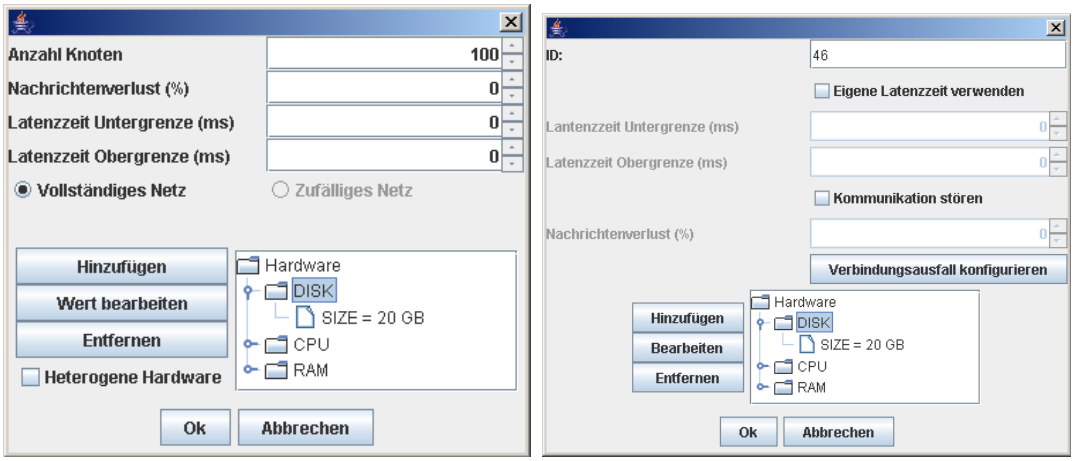


Abbildung 4.2: Dialoge zum Erstellen des Netzwerks und zum Editieren eines Knotens

- **Nachrichtenverlust**
  Die Wahrscheinlichkeit in Prozent, dass eine Nachricht verloren geht

- **Latenzzeit Unter- und Obergrenze**
  Der Bereich aus dem zufällig ein Wert für die Latenzzeit einer Nachricht ermittelt wird.

- **Vollständiges Netz**
  Generiert ein voll vernetztes Netzwerk. Jeder Knoten hat eine Verbindung zu jedem anderen Knoten.

Im unteren Bereich des Dialogs kann die Hardware als Vorlage für die Knoten eingestellt werden. Wird ein Punkt markiert, so ändert sich die Funktionsweise der Schaltflächen. Bei Hardware können Ressourcen und bei Ressourcen Werte hinzugefügt werden. Werte schließlich können editiert werden. Die Auswahl hängt davon ab, was in der *Default Hardware* Datei angegeben ist. Die ausgewählte Hardware dient als Grundlage für die Knoten, die generiert werden. Ist der Punkt *Heterogene Hardware* ausgewählt, so variieren die Werte von Knoten zu Knoten ansonsten hat jeder die gleiche Hardware.

Wurde der Dialog bestätigt, so wird das Netzwerk und die Knoten generiert. Die Darstellung verändert sich und zeigt die Übersicht der Knoten. Jeder Knoten hat einen eindeutigen Bezeichner(„ID“) erhalten, der auf seinem Label zu sehen ist. Diese Label können mit der Maus angeklickt werden. Es öffnet sich ein Dialog, in dem Eigenschaften des Knotens editiert werden können. Ein Beispiel für einen solchen Dialog ist Abbildung 4.2.

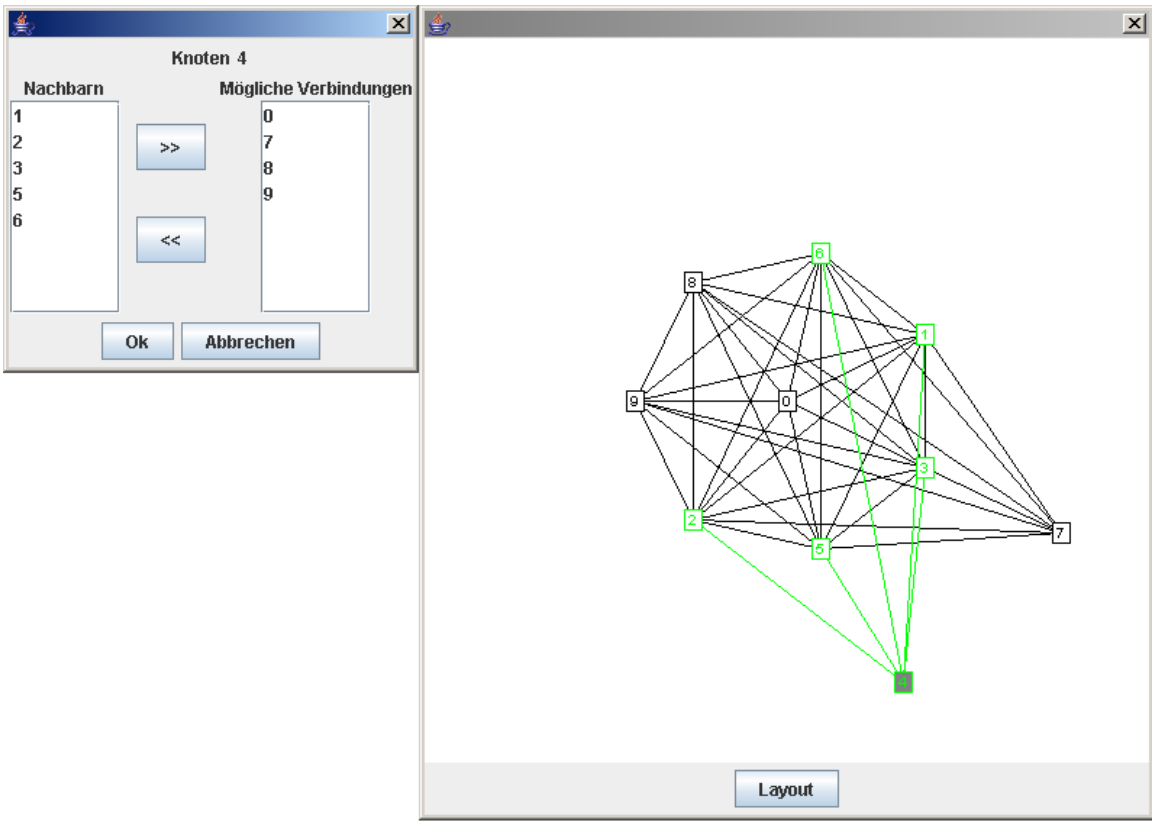


Abbildung 4.3: Editieren der Netzwerkverbindungen

Jeder Bezeichner lässt sich zu diesem Zeitpunkt noch verändern. Dieser wird in in den Logmeldungen verwendet. Erhält ein spezieller Knoten eine andere Bezeichnung, wie etwa „A“ so lässt sich dieser Knoten sehr einfach wiederfinden. Das dient zur Untersuchung von Sonderfällen. Wird beispielsweise der Nachrichtenverlust eines Knotens sehr hoch eingestellt, so kann im Log die Auswirkungen einfach verfolgt werden.

Für das gesamte Netz kann der Nachrichtenverlust und die Latenzzeit eingestellt werden. Diese Werte gibt es nochmal für einen Knoten. Damit kann ein unzuverlässiger Knoten simuliert werden. Zudem lassen sich einzelne Verbindungen stören. Wird die Schaltfläche *Verbindungsausfall konfigurieren* betätigt, dann kann im folgenden Dialog eine existierende Verbindung ausgewählt werden und eine eigene Verlustwahrscheinlichkeit eingetragen werden.

Als letztes kann noch die Hardware eines Knotens spezifiziert werden. Die vorhandene Hardware entspricht den Vorgaben, welche im Dialog zur Erstellung eines Simulationslaufs gemacht wurden. Soll ein Knoten unterschiedliche Ressourcen und Werte haben, so können die Veränderungen hier durchgeführt werden. Kehrt man zurück zum Hauptfenster, kann über die Schaltfläche *Network View* eine Darstellung der Netzwerkverbindungen wie in Abbildung 4.3 eingesehen werden. Jeder Knoten hat

eine grafische Repräsentation mit seinem Bezeichner und jede Linie stellt eine bestehende Verbindung zu einem anderen Knoten dar. Ist der Mauszeiger über einem Knoten, so werden seine Nachbarn farblich markiert. Wird ein Knoten angeklickt, so kann in einem Dialog die Nachbarn des Knotens spezifiziert werden, was Änderungen in den Verbindungen des Netzwerks bewirkt. Die Position jedes Knotens lässt sich verändern, in dem er mit der Maus an eine andere Stelle gezogen wird. Das Netzwerk kann wieder automatisch angeordnet werden, wenn die Schaltfläche *Layout* betätigt wird.

Für die Darstellung wird das *Prefuse Toolkit* [HCL05] verwendet, das darauf spezialisert ist, verknüpfte Informationen darzustellen. Die Darstellung eines voll vernetzten Peer to Peer Netzwerks leidet durch die vielen Verbindungen, weshalb es zu Leistungsproblemen ab etwa 100 Knoten kommt.

Sind alle Einstellungen für die Knoten vorgenommen, kann über die Schaltfläche *Start* der Startvorgang eingeleitet werden. Ein letzter Dialog erscheint, in dem ein Startknoten ausgewählt wird, der die Konfiguration verarbeiten und verteilen soll. Für jeden Simulationslauf wird eine Konfiguration benötigt, wobei entweder eine existierende in einer Datei verwendet werden kann oder es wird eine passende generiert. Die generierte Konfiguration wird in einer Datei gespeichert, damit sie auch nach dem Durchlauf verfügbar bleibt. Es stehen zwei Konfigurationsgeneratoren zur Wahl, die noch in Abschnitt 6.1 im Detail erklärt werden. Dort werden ebenfalls die Parameter und ihre Auswirkung erklärt weshalb hier nicht weiter darauf eingegangen wird. Wird die Auswahl bestätigt, so startet der Simulator den Ablauf. Der gewählte Startknoten lädt die Konfigurationsdatei, parst den Inhalt und validiert diesen. Wurden keine Fehler gefunden, wird die Konfiguration verteilt und die Knoten beginnen mit der Aushandlung.

Der Status jedes Knotens wird mit der Farbe des Labels dargestellt. Während der Aushandlung wird ein Farbverlauf von Dunkelblau bis Weiß verwendet, um die Prozentzahl der erfüllten Jobs wiederzugeben. Eine erfüllte Konfiguration wird mit einer grünen Farbe dargestellt. Um eine Rückmeldung bei Problemen zu geben, werden ebenfalls Farben verwendet. Wird beispielsweise ein Job nicht erbracht und es ist kein Dienstleister bekannt, so wechselt die Farbe zu Gelb während versucht wird, das Problem zu beheben. Scheitert dies, so ist ein Fehler aufgetreten die allesamt mit Rot dargestellt werden.

Ein Simulationslauf kann mit der Schaltfläche *Stop* abgebrochen und mit *Pause* angehalten werden. Für letzteren Fall kann der Lauf mit dem Betätigen von *Play* fortgesetzt werden. Wird auf einen Knoten geklickt, so können einige seiner momentanen Werte eingesehen werden. Dazu gehören neben der Konfiguration und den eingetragenen Dienstleistern auch die Joblisten sowie die Hardware nebst Auslastung.

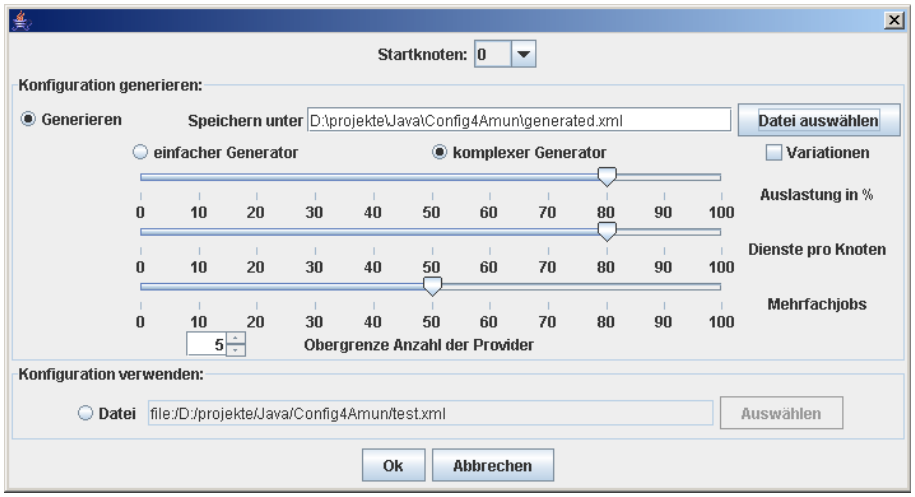


Abbildung 4.4: Auswahl der Konfiguration

Am Ende eines Simulationslaufs wird nach einer Zeitspanne ein Snapshot gemacht, in dem überprüft wird, ob die Einträge in den Konfigurationen der Knoten konsistent sind. Zur manuellen Kontrolle des Ergebnisses dient der Unterpunkt *Ergebnis prüfen* im Menüpunkt *Tools*. Werden Inkonsistenzen festgestellt, wird genau ausgegeben, bei welchen Knoten und Jobs diese gefunden wurde und der gefundene Unterschied. Ansonsten wird nur bestätigt, dass der Test bestanden wurde. Die *Statistik* im gleichen Menü zeigt die Anzahl der verschickten Nachrichten sowie weitere Werte wie beispielsweise die Anzahl der Jobs. Diese Werte dienen der Auswertung die in Kapitel 6 beschrieben ist.

4.3 Protokollierung

Überaus wichtig für die Entwicklung und Überwachung des Algorithmus ist eine Nachvollziehbarkeit der Aktionen. Dazu müssen verschiedene Ereignisse in einem Log protokolliert werden. Dafür findet in dieser Arbeit *Log4j* [log05] Verwendung, ein leistungsfähiges und bewährtes Framework für das Protokollieren.

Da es zu einer Menge an Log-Einträgen kommt, sollte für die Speicherung eine Datenbank verwendet werden anstelle einer Textdatei. Die Datenbank bietet mit SQL eine Abfragesprache mit der die Suche und Filterung der Daten einfach ist, während das für eine Textdatei aufwändig mit externen Werkzeugen gemacht werden muss.

Log4j bietet keinen besonders guten Appender[1] für die Datenbank, stattdessen wird ein alternativer JDBCAppender [Man05] verwendet, der leistungsfähiger und zugleich

[1] In der Nachfolge Version 1.30 soll es einen leistungsfähigen Ersatz geben

flexibler ist.

Um *Log4j* zu Verwenden, werden in der *dblogger.properties* Datei die Logger und Appender spezifiziert. Alle Einstellungen sind in der Dokumentation zu *Log4j* [log05] und dem JDBCAppender [Man05] beschrieben. Auf jeden Fall notwendig sind die Verbindungsdaten zur Datenbank, die Angabe des JDBC Treibers und auch die Angabe des `sqlhandlers`. Damit werden die Log-Einträge in die Datenbank geschrieben und dafür wird die Klasse `MySqlHandler` im Paket `simulation.util.db` verwendet.

Der Einsatz des Logging Frameworks bietet einige Vorteile. Mit Hilfe des Loglevels lässt sich einstellen, dass nur wichtige Einträge protokolliert werden, damit die Übersichtlichkeit nicht verloren geht. Alle versandten Nachrichten werden protokolliert, so dass deren Anzahl über das Log bestimmt werden kann und sich die Werte der Zähler bestätigen lassen. Damit sich der Ablauf des Algorithmus nachvollziehen lässt, fügt jeder Knoten seinen Logmeldungen seine ID hinzu. Das ermöglicht eine Projektion auf den Ablauf aus Sicht eines Knotens, was besonders bei besonderen Ereignissen, wie beispielsweise verlorenen Nachrichten wichtig ist.

4.4 Batchmodus

Neben dem grafischen Modus kann der Simulator auch in einem Batchmodus gestartet werden. Die Einstellungen für einen Simulationslauf müssen in einer Datei vermerkt sein, damit das Programm ausgeführt werden kann. Diese Datei kann explizit angegeben werden oder das Programm versucht die `settings.xml` zu laden, die von der Oberfläche generiert wird.

Wird die Hauptklasse `simulation.Main` ohne Parameter aufgerufen so erscheint eine Übersicht der Kommandozeilenparameter.

- **-batch**
  Startet im Batchmodus. Hierfür müssen sich die Einstellungen in einer Datei befinden.
- **-useFile**
  Die Datei in der die Einstellungen für den Simulationslauf vermerkt sind.
- **-simulator**
  Startet im grafischen Modus. Die Parameter `-batch` und `-simulator` schließen sich gegenseitig aus.

Eine beispielhafte Darstellung ist im Anhang A zu finden.

Der Ablauf einer Simulation im Batchmodus kann nur über die Datenbank vernünftig verfolgt werden. Es wird auf jeden Fall für die Terminierung des Programms gesorgt. Am Ende steht ein Snapshot der das Ergebnis festhält und das System kontrolliert beendet.

4.5 XML Serialisierung

Es gibt eine Reihe Einstellungsmöglichkeiten für den Simulator. Diese können zwar über die Oberfläche vorgenommen werden, aber gleichzeitig sollen diese für den Batchbetrieb ebenfalls möglich sein. Bei der großen Anzahl an Parametern ist es sinnvoll, diese in einer Datei festzulegen.

Dafür bieten sich entweder Property Dateien an oder eine eigene Definition. XML bietet sich auch hier an, da es aussagekräftig und erweiterbar ist. Prinzipiell würden die Einstellungen in der Datei geparst und dann als Werte in ein Objekt gesetzt werden. Eine deutlich einfachere Lösung ist die XML Serialiserung, mit der die Objekte als XML gespeichert und wieder ausgelesen werden können. Es existieren Objekte, die nur für die Konfiguration vorgesehen sind und es ermöglichen, den Simulator entweder über die Oberfläche oder per Datei zu konfigurieren.

Ein Framework, das dies leistet ist *Castor* [cas05], welches in dieser Arbeit eingesetzt wird. Es bietet einen einfachen Mechanismus, um auch komplexe Objekte serialisieren zu können. Die Objekte müssen dazu nur der Java Bean Konvention folgen. Castor verwendet die Java Reflection API um die Serialisierung durchzuführen, was bei einfachen Objekten ausreicht. Für komplexere Objekte werden Übersetzungsregeln („Mapping“) notwendig, um die Abbildung von XML auf das Objekt und umgekehrt zu spezifizieren. Das ist notwendig, um Namenskonflikte aufzulösen und auch um Einträge in Klassen des Java Collection Frameworks, wie etwa der `HashMap`, richtig zu serialisieren.

Diese Regeln befinden sich in der so genannten Mapping Datei, die für den Simulator benötigt wird. Damit nicht alle Einträge per Hand geschrieben werden müssen, existiert im package `build` die Klasse `CreateMappings`, welche mit einem Tool von Castor für die Objekte eine Mapping Datei generiert. Diese sollte zuerst geprüft werden, ob damit auch eine Serialisierung durchführbar ist, was ebenfalls mit `CreateMappings` machbar ist, welches dann verschiedene Beispiel-Einstellungen verwendet. Hier muss häufig per Hand nachgebessert werden, die entsprechenden Regeln finden sich auf der *Castor* Homepage [cas05].

5 Selbstkonfiguration durch Kooperation

5.1 Definition

Eine Selbstkonfiguration nach [KC03] beschreibt Richtlinien, die ein Ziel festlegen, das erreicht werden soll ohne genau zu spezifizieren, wie es erreicht werden soll. Die Idee dahinter ist, dass ein Administrator das Ziel vorgibt und das System es selbstständig umsetzt. Zusätzlich sollen neue Komponenten, die dem System hinzugefügt werden, sich nahtlos einfügen und sich entsprechend der Gegebenheiten selbst konfigurieren. Der Rest des Systems kann die neue Funktionalität der eigenen hinzufügen und nutzen.

In Abschnitt 3.4 wurde die Konfiguration erläutert, welche die Richtlinien beschreibt und die Grundlage für die Selbstkonfiguration ist. Diese soll durch Kooperation der Knoten erreicht werden. Dazu wird ein kooperativer Algorithmus verwendet, dessen Merkmal es ist, dass die Handlungen von allen Beteiligten zum Erfolg des Gesamtsystems beitragen. Zum Verständnis wird der Ablauf zuerst mit folgendem, abstrakten Beispiel beschrieben.

Eine Fabrik ist neu errichtet worden und alle Stellen müssen besetzt werden. Erst wenn alle Stellen mit dem qualifiziertesten Personal besetzt ist, kann die Fabrik optimal produzieren. Es gibt Stellen, die können nur von einer Person besetzt werden, andere dagegen können mehrfach besetzt werden. Es gibt eine Anzahl an Bewerbern, von denen manche auch verschiedene Stellen übernehmen können. So kann der Gabelstaplerfahrer auch am Fließband arbeiten, aber er ist weitaus besser im Staplerfahren. Der Fließbandarbeiter kann dagegen den Job des Gabelstaplerfahrers nicht übernehmen, wenn er den notwendigen Führerschein nicht besitzt. Gibt es einen hohen Bedarf an Fahrern, dann ist es vielleicht sinnvoll, dass jeder der einen Gabelstaplerführerschein hat, zuerst einen Job als Fahrer übernimmt.

Alle Stellen werden an einem Nachmittag vergeben, also versammeln sich alle Bewerber auf dem Parkplatz vor der Fabrik. Jeder Bewerber erhält eine Liste mit den Stellen und markiert sich diejenigen, für die er qualifiziert ist und sortiert sie so, dass

die besten für ihn oben stehen. Danach kann jeder Bewerber laut rufen, welche Stelle er übernimmt. Ist jemand der Ansicht das er es besser kann, so überstimmt er ihn. Der Beste bekommt den Job und alle streichen den Job von ihrer Liste und wenden sich dem nächsten zu. Gibt es mehr Bewerber als Jobs, so gehen manche leer aus, gibt es dagegen weniger, müssen manche mehr als einen Job machen. Es kann auch sein, dass es Stellen gibt, die kein Bewerber besetzen kann. Ist für den Posten des Fabrikchefs ein Studium erforderlich und niemand kann es aufweisen, so kann die Stelle nicht besetzt werden und die Fabrik kann nicht produzieren.

5.1.1 Abbildung auf das Modell

Überträgt man das Beispiel auf das Modell, so stellt die Fabrik die Applikation dar und die Stellen werden von den Diensten repräsentiert. Im Modell fungieren die Knoten als Bewerber die versuchen alle Dienste zu erbringen, damit die Applikation funktionieren kann.

Abhängig von der vorhandenen Hardware und Software kann ein Knoten verschiedene Dienste erbringen, besonders leistungsstarke Rechner auch mehrere. Was dabei an Anforderungen vorhanden sein muss, ist in der Konfigurationsbeschreibung festgelegt, welche mit der Liste der Stellen im Beispiel vergleichbar ist.

Es gibt kein Treffen zu einem bestimmten Zeitpunkt sondern ein Knoten beginnt mit der Bewerbung, sobald er die Konfigurationsbeschreibung erhalten hat. Zuerst werden alle Dienste auf Ausführbarkeit geprüft und eine Dienstgüte berechnet, nach der die Liste sortiert wird. Ausführbarkeit bedeutet, dass alle Anforderungen an einen Dienst erfüllt werden. Im Beispiel konnte der Job des Gabelstaplerfahrers nur mit einem Gabelstaplerführerschein erbracht werden. Jeder Bewerber konnte sich die für ihn besten Stellen auswählen, dafür gibt es im Modell eine Funktion zur Berechnung der Dienstgüte, die diese Bewertung liefert.

Im Beispiel beginnen die Bewerber, sich durch Rufen um die Stellen zu bewerben. Im Modell verschicken die Knoten Broadcast Nachrichten, um die Ausführung eines Dienstes bekannt zu geben.

Sind alle Stellen vergeben, so kann die Arbeit in der Fabrik aufgenommen werden. Jeder Bewerber kann das erkennen, wenn alle Stellen besetzt sind. Ebenso verhält es sich im Modell. Hat jeder Job die notwendige Anzahl an Dienstleistern, so ist die Konfiguration erfüllt. In einem letzten Schritt muss allen Knoten diese Information mitgeteilt werden. Unter Umständen ist in der Kommunikation ein Fehler aufgetreten und damit wird erreicht, dass jeder Knoten auf dem aktuellem Stand ist und auch die gleichen Informationen hat.

Daraus lassen sich drei Phasen des Algorithmus ableiten:

1. **Konfigurationsbeschreibung und Verteilung**
   Die Konfiguration wird im Netzwerk verteilt und jeder Knoten erarbeitet für sich, welche Dienste er ausführen kann und wie gut sie erbracht werden können.

2. **Aushandlung**
   Die Knoten handeln untereinander aus, wer welchen Job übernimmt. Das wird so lange durchgeführt, bis die Konfiguration erfüllt ist.

3. **Verifizierung**
   Es wird geprüft ob die gesammelten Information der Aushandlung übereinstimmen und gegebenenfalls Maßnahmen zur Korrektur durchgeführt.

5.2 Konfiguration und Verteilung

Am Anfang sind die Knoten in einem Zustand, in dem sie nur auf eingehende Nachrichten lauschen und selbst nicht aktiv werden. Die Konfiguration liegt im XML-Format vor und ist über das Netzwerk erreichbar. Der Administrator erteilt einem Knoten aus dem Netzwerk die Aufgabe über eine URL die Konfigurationsdatei zu laden. Diese wird automatisch geparst, anhand eines Schemas verifiziert und im Anschluss daran an die anderen Knoten per Broadcast verteilt.

Erhält ein Knoten die Konfiguration, so beginnt die Abarbeitung, welche als Pseudocode in Algorithmus 1 dargestellt ist.

Zuerst werden die `Constraints` aus der Konfiguration geprüft. Treffen die Voraussetzungen zu, dann wird der entsprechende Dienst zur Ausführung markiert und die vorhandenen Ressourcen werden um die in den Anforderungen angegebenen Werte verringert. Die Constraints spielen für den Rest des Algorithmus keine Rolle mehr, da sie keine Abstimmung im Rahmen der Aushandlung mit den anderen Knoten benötigen.

Der nächste Schritt besteht darin, die vorhandenen Jobs zu prüfen. Auch hier wird zuerst eine Prüfung durchgeführt, ob der Dienst ausführbar ist und falls nicht, so wird dieser in einer speziellen Liste gemerkt. Damit lässt sich später herausfinden, ob die Konfiguration erfüllbar ist oder nicht, worauf noch später eingegangen wird. Zu jedem ausführbaren Dienst wird eine Quality of Service berechnet. Als Grundlage dafür dienen die maximal verfügbaren Ressourcen abzüglich der Constraints. Dieser Wert drückt somit aus, wie gut der Dienst ausgeführt werden würde, wenn sonst kein anderer Dienst laufen würde. Alle machbaren Dienste werden sortiert nach ihrer Quality of Service in einer Liste gehalten, der „Jobliste“.

Algorithmus 1 Auswahl und Bewertung der Jobs

```
constraints ← configuration.getConstraints();
jobs ← configuration.getJobs();
joblist ← new List();
undoableJobs ← new List();
for all constraint c in constraints do
  if canBeProvided(c) then
    provide(c.getServices());
    reduceAvailabeCapacity(c.getRequirements());
  end if
end for
for all job j in jobs do
  if canBeProvided(j) then
    qos ← calculateQualityOfService(j);
    joblist.add((qos, j));
  else
    undoableJobs.add(j);
  end if
end for
sort(joblist);
```

Sobald dieser erste Schritt abgeschlossen ist, hat der Knoten eine Liste mit den ausführbaren Jobs. Dank der Sortierung nach der Dienstgüte, kann immer der Dienst ausgewählt werden, der vom Knoten theoretisch am besten ausgeführt werden kann. Damit kann die Aushandlung beginnen.

5.3 Aushandlung

Jeder Knoten hat einen aktiven und einen passiven Zustand, der immer im Wechsel durchlaufen wird. Im passiven Zustand wird nur „gelauscht“ und Nachrichten gesammelt. Im aktiven Zustand werden die erhaltenen Nachrichten verarbeitet und danach eine Entscheidung gefällt, was als nächstes zu tun ist.

5.3.1 Erbringung von Diensten

Die Entscheidung hängt davon ab, ob noch Dienste zu erbringen sind beziehungsweise besser erbracht werden können. In diesem Fall wird ein entsprechender Dienst ausgewählt und per Broadcast den anderen Knoten die Information mitgeteilt.

Algorithmus 2 Entscheidung der Erbringung eines Jobs

```
while ! empty(joblist) do
  (qos, job) ← joblist.removeFirst();
  quality ← calculateQualityOfService(job);
  if job.isOpen() OR job.canBeProvidedBetter(quality) then
    provide(job);
    notifyOthers(job, quality);
    return job;
  end if
end while
```

Die Entscheidungsfindung wird in Algorithmus 2 dargestellt. Die Jobliste wird abgearbeitet, allerdings muss darauf geachtet werden, nur einen Dienst zu erbringen und dann den anderen Knoten die Chance zu geben, selbst zum Zuge zu kommen. Damit soll eine bessere Verteilung erreicht werden.

Es wird versucht einen Job zu finden, der entweder noch erbracht werden muss oder verbessert werden kann. Dazu wird die Liste in einer Schleife abgearbeitet und ein entsprechender Eintrag zurückgeliefert. Zuerst wird der Eintrag aus der sortierten Liste entnommen und eine neue Dienstgüte berechnet. Zwar wurde diese bei der Abarbeitung der Konfiguration bereits einmal berechnet, allerdings können sich die

vorhandenen Ressourcen bei jedem Durchlauf des Algorithmus ändern. Erbringt der Knoten Dienste, so werden diese geringer, wird er von einem anderen Knoten überschrieben, so wird wieder Kapazität frei.

Im nächsten Schritt wird geprüft, ob der Dienst noch erbracht werden muss, oder ob er verbessert werden kann. Jeder Job enthält eine Liste mit den Dienstleistenden, also den Knoten, welche den Dienst erbringen. Ersteres ist der Fall, wenn in der Liste noch ein Platz frei ist.

Sollte ein Knoten einen Dienst übernehmen, bedeutet das noch lange nicht, dass ein anderer Knoten diesen Dienst nicht besser erbringen kann. In diesem Fall wird die Quality of Service des schlechtesten Eintrags der Liste mit dem errechneten Wert verglichen und wenn der Knoten besser ist, überschreibt er den Eintrag. Damit soll eine möglichst optimale Verteilung erreicht werden. In der Auswertung hat sich gezeigt, dass an dieser Stelle noch optimiert werden muss, da es mitunter zu einer erheblichen Anzahl Überschreibungen kommt.

Wird ein Dienst erbracht, so wird dieser zur Ausführung markiert und die Anforderungen werden von den vorhanden Ressourcen abgezogen. Der Knoten setzt die anderen mit einer Broadcast Nachricht davon in Kenntnis, dass er diesen Dienst erbringt und teilt ihnen seine ID sowie die Dienstgüte mit.

5.3.2 Konflikte bei der Aushandlung

Soll ein Job von mehr als einem Dienstleister erbracht werden, kann es vorkommen, dass ein Knoten diesen Job mehrfach erbringt. Da unter Umständen die Quality of Service ebenfalls den gleichen Wert haben kann, muss eine weitere Information hinzukommen, um diese Informationen unterscheiden zu können. Dazu benötigt man eine weitere ID, wobei es allerdings völlig ausreicht, wenn diese nur für den Dienstleister eindeutig ist[1].

Idealerweise erbringt immer ein Knoten einen Job, allerdings passiert es in der Realität häufig, das zwei Knoten den gleichen Job zur gleichen Zeit erbringen wollen. Das führt nur dann zu einem Problem, wenn ein Job nur einen Provider vorsieht oder nur ein Platz in der Dienstleisterliste frei ist. Dann kann es zu einem Konflikt zwischen zwei Knoten kommen. Soll ein Dienst erbracht werden, so wird das derjenige tun, welcher die bessere Dienstgüte aufweist. Unter der Annahme, das keine Fehler bei der Kommunikation auftreten, erhält jeder Knoten die gleiche Information und kommt so zum gleichen Ergebnis. Damit kann ein Konflikt nur eintreten, wenn beide Dienstleister die gleiche Dienstgüte aufweisen.

[1] In der Simulation wird dafür die Nachrichten ID verwendet, was den Vorteil hat, die Information einer Nachricht zuordnen zu können

Um eine Lösung zu finden, könnten sich beide Dienstleister untereinander einigen und das Ergebnis bekannt machen. Dazu würden sich beide weitere Informationen schicken, wie etwa die durchschnittliche Auslastung oder Anzahl der Dienste die bereits erbracht werden. Eine Einigung muss auf jeden Fall erzielt werden, zur Not mittels eines Zufallwertes. Allerdings kostet eine solche Einigung Zeit, mindestens zwei Unicast Nachrichten und eine Broadcast Nachricht. Diese Verzögerung und die Nachrichten können gespart werden, wenn man die Informationen zur Einigung bereits bei der ersten Nachricht mitschickt. In diesem Fall muss jeder Knoten anhand der Informationen entscheiden können, welcher Dienstleister den Job erbringt.

Da sich Nachrichten überholen können oder in bestimmten Regionen des Netzwerks verzögert ankommen, darf die Eingangsreihenfolge der Nachrichten für eine Entscheidung nicht betrachtet werden, da sie nicht global gleich ist. Damit würde sich jeder Knoten anders entscheiden, was letztlich zu Inkonsistenzen führt.

Die Dienstgüte alleine reicht ebenfalls nicht aus, da es sein kann, dass zwei Knoten den Job mit der gleichen Güte erbringen können. Bei gleicher Hardware ist das sogar eher die Regel. Um also eine Entscheidung zu ermöglichen, werden noch zusätzliche Informationen verwendet:

1. **Auslastung der Knoten**
   Der Knoten der insgesamt am geringsten ausgelastet ist, erbringt den Dienst.

2. **Anzahl der bereits zum Starten markierter Dienste**
   Mehr Dienste verursachen auch mehr Overhead, weshalb der Knoten mit weniger Diensten die Ausführung übernimmt.

3. **Größe der Jobliste mit noch machbaren Dienste**
   Der Knoten mit einer längeren Jobliste kann potentiell andere Dienste erbringen und sollte dem anderen den Vorzug gewähren. Als weitere Entscheidung kann auch die Liste der optionalen Dienste dienen.

4. **Zufallszahl**
   Sollte bis hierhin keine Entscheidung gefallen sein, so wird eine Zufallszahl herangezogen, welche einmalig bei der Erzeugung der Nachricht erstellt wird.

5. **ID der Knoten**
   Als letzte Entscheidungsinstanz wird die ID der Knoten verwendet, da diese eindeutig sind und somit keine Gleichheit bestehen.

Die Entscheidung folgt der obigen Reihenfolge. Bei Gleichheit wird die nächste Stufe geprüft bis zur letzten, die eine Entscheidung garantiert.

Die zusätzlichen Informationen benötigen nicht viel Platz, da es nur vier Integer sind. Damit existieren automatisch keine Konflikte und es gibt einen nützlichen Nebeneffekt, denn es lässt sich jederzeit rekonstruieren, welche Dienstleister einen Job erbringen. Diese Eigenschaft wird später nützlich werden, wenn eine Konfiguration korrigiert werden muss.

5.3.3 Abschluß der Aushandlung

Jeder Job enthält die Liste seiner Dienstleister. In jeder aktiven Phase wird geprüft, ob alle Jobs erbracht worden sind. Ist dies der Fall, so wird nach einer festgelegten Wartezeit mit einer Broadcast Nachricht den anderen Knoten mitgeteilt, dass die Konfiguration erfüllt ist. Die Wartezeit ist notwendig, da noch Nachrichten unterwegs sein können, welche möglicherweise eine Verbesserung enthalten. Würde sofort eine Nachricht verschickt werden, würde das eine potentielle Inkonsistenz beinhalten, da andere Knoten die Verbesserung vielleicht eingetragen haben und die Konfiguration für inkorrekt halten.

Es gibt einen weiteren Fall, der berücksichtigt werden muss. Dazu wird nochmal ein Beispiel betrachtet.

Niemand hat sich um eine Stelle beworben, was daran liegen kann, dass niemand die notwendige Qualifikation hat. Nachdem eine Zeit verstrichen ist, in der niemand mehr gerufen hat, wird sich jeder Bewerber der die Stelle nicht selbst übernehmen kann wundern, ob er vielleicht einfach nur die Bewerbung überhört hat und nachfragen. Die Frage gestaltet sich ebenfalls als ein Rufen, so dass jeder der zuhört auf seiner Liste nachsehen kann, ob die Stelle besetzt ist und falls ja, dem fragenden Bewerber die Information mitteilen kann.

Im Modell wird ähnlich wie im Beispiel verfahren. Der Knoten kann den Dienst selber nicht ausführen, weil die Anforderungen nicht erfüllt werden können. Also wird nach einem Timeout in der keine Nachricht eingegangen ist, per Broadcast bei den anderen Knoten nachgefragt, wer die Dienstleistenden sind.

Jeder Knoten der die Anfrage erhält, überprüft seine Liste mit Dienstleistern. Finden sich in der Liste Einträge zu diesem Dienst, so wird er dem Sender antworten. Ansonsten wird er still bleiben und lauschen, denn offensichtlich benötigt er selbst die Information.

Bleibt auf die ursprüngliche Anfrage eine Antwort aus, so kann entweder kein Knoten diesen Dienst erbringen oder niemand hat die Frage erhalten. Deswegen prüft der fragende Knoten nach einer Zeitspanne, ob die Verbindung unterbrochen wurde. Dazu verschickt er eine Broadcast Nachricht auf die jeder Knoten garantiert antworten wird.

Gibt es keine Antwort, dann ist der Knoten isoliert, andernfalls ist die Konfiguration nicht erfüllbar und die Applikation kann nicht gestartet werden. In beiden Fällen muss das System eine Rückmeldung an den Administrator geben.

5.4 Verifizierung

Zu diesem Zeitpunkt existiert eine Konfiguration die erfüllbar ist. Die Applikation ließe sich damit tatsächlich ausführen. Aber es ist nicht gewährleistet, dass diese Information auf jedem Knoten konsistent ist. Um zu Verhindern, das manche Dienste doppelt oder gar nicht gestartet werden, ist eine Verifikation notwendig.

Die Nachricht zur Meldung der erfüllten Konfiguration enthält auch die Informationen der Dienstleister, die der Sender gesammelt hat. Anhand dieser kann jeder Empfänger vergleichen, ob die Konfiguration mit seinen Informationen übereinstimmt. Falls Inkonsistenzen erkannt werden, wird eine optimale Lösung aus der Dienstgüte der Dienstleister errechnet. Das ist nur möglich, da jederzeit die Entscheidung nachvollzogen werden kann, welcher Knoten einen Dienst erbringt.

Ergibt sich, dass der Empfänger eine schlechtere Lösung hatte, so übernimmt er diese Werte. Beispielsweise würde dieser Fall eintreten, wenn der Empfänger eine Überschreibung nicht mitbekommen hätte. Mittels der Information in der Nachricht kann aber diese Überschreibung wiederhergestellt werden.

Anders verhält es sich, wenn der Sender die schlechtere Lösung hatte. In diesem Fall gibt es einen Konflikt der gelöst werden muss. Der Empfänger schickt in diesem Fall eine Konfliktnachricht, in der die Fehler und ihre Lösung enthalten sind. Da unter Umständen diese Information ebenfalls bei anderen Knoten fehlt, wird das per Broadcast gesendet. Damit hat jeder Knoten der diese Nachricht empfängt die Möglichkeit diesen Fehler direkt zu beheben.

Gibt es keine Konflikt Nachrichten von anderen, so wird optimistisch nach einer Zeitspanne angenommen, dass die Konfiguration korrekt ist. Im Idealfall wird also nur eine Nachricht benötigt um das Ergebnis zu verifizieren.

6 Auswertung

Um genügend Daten für eine Auswertung des Algorithmus zu sammeln, is eine hohe Anzahl an Durchläufen der Simulation notwendig. Die Oberfläche der Simulation ist darauf ausgerichtet, einen grafischen Ablauf des Algorithmus darzustellen und eignet sich dafür eher weniger.

Das System lässt sich aber komplett ohne grafische Darstellung in einem Kommandozeilen Betriebsmodus starten und alle notwendigen Einstellungen für einen Ablauf werden in XML Dateien definiert. Es war ursprünglich gedacht, eine Konfiguration für die Testläufe zu erstellen. Das hätte bedeutet, für jeden Testfall eine Datei mit passenden Einträgen zu erzeugen. Dabei besteht das Risiko, mit einer besonders günstig oder ungünstig gewählten Konfiguration, die Auswertung zu beeinflussen. Aus diesen Gründen war es nahe liegend, eine neue Konfiguration für jeden Testlauf zu generieren.

6.1 Generieren einer Konfiguration

Um einen Generator für eine Konfiguration zu erstellen, muss beachtet werden, dass dieser eine korrekte und erfüllbare Konfiguration erzeugt. Korrektheit ist dadurch gewährleistet, dass die Ausgabe in einem gültigen Format geschrieben wird und auch gegen das Schema validiert werden kann.

Eine erfüllbare Konfiguration lässt sich erreichen, wenn nur Dienste existieren für die auch die passende Hardware auf den Knoten vorhanden ist. Ansonsten würde die Simulation einen Fehler melden, da Jobs existieren, die nicht erfüllbar sind.

Es ist leichter, wenn die Konfiguration generiert wird, nachdem das simulierte Netzwerk mit seinen Knoten aufgebaut wurde. Dann reicht es, die Daten der vorhandenen Hardware auf jeden Knoten zu notieren und aus diesen Angaben die entsprechenden Dienste zu erzeugen.

Anforderungen der verschiedenen Testläufe sind, das Verhalten des kooperativen Algorithmus unter verschiedener Auslastung der Knoten (`workload`) zu simulieren. Da im ersten Schritt die gesamte Hardware durchlaufen wird, können auch gleichzeitig

die Werte der Ressourcen aufsummiert werden. Aus der entstandenen Gesamtsumme kann die notwendige Auslastung berechnet und daraus angepasste Anforderungen für die Dienste erzeugt werden.

Zu diesem Zweck existieren spezielle Datentypen, nämlich `CountingRessource` und `CountingValue`. Erstere Klasse dient dazu die Anzahl und die Werte einer Ressourcenart festzuhalten. Dazu enthält sie eine entsprechende Menge an `CountingValue`. Letztere dienen hauptsächlich dazu, für die vorhandenen Werte die Summe zu ermitteln. Der Ablauf zur Ermittlung der Ressourcen und Werte ist in Algorithmus 3 dargestellt.

Algorithmus 3 Zählen der Ressourcen

```
Require: nodes
  countingRessources ← newMap();
  for all Node node in nodes do
    allRessources ← node.getRessources();
    for all Ressource r in allRessources do
      wrappingCounter = countingRessources.get(r.getName());
      if wrappingCounter = null then
        wrappingCounter = newCountingRessource(r);
        countingRessources.put(r.getName(), wrappingCounter);
      else
        wrappingCounter.add(r);
      end if
    end for
  end for
```

Die Voraussetzung sind die bereits erzeugten Knoten, die in `nodes` übergeben werden. Es wird eine Datenstruktur benötigt, welche eine Zuordnung des Ressourcen Typs und der `CountingRessource` enthält. Für ersteres existiert ein eindeutiger Name, welcher die Ressourcen gleichen Typs identifiziert. Bei der Berechnung werden die Ressourcen von jedem Knoten ausgelesen. Falls noch keine entsprechende `CountingRessource` existiert, so wird eine neue erzeugt und in der Datenstruktur abgelegt. Ansonsten wird der Wert der Ressource hinzu addiert. Nachdem alle Knoten abgearbeitet sind, kann die eigentliche Generierung beginnen, wobei die zuvor erzeugte Datenstruktur benötigt wird.

6.1.1 Einfacher Generator

Die erste Implementierung eines Generators basiert auf der einfachen Idee, für jeden Knoten einen Job zu erstellen, der alle Ressourcen als Anforderungen hat. Da aber

nicht garantiert ist, dass alle Knoten auch die selben Ressourcen haben, werden für einzelne Ressourcen Constraints erstellt.

Algorithmus 4 Einfacher Konfigurations Generator

```
Require: nodes, numberNodes, workload, countingRessource
  id ← 1;
  constraintList ← newList();
  jobList ← newList();
  for all CountingRessource cr in countingRessource do
    counted ← cr.getCounted();
    ressource ← cr.createRessource();
    for all CountingValue cv in cr.getCountingValues() do
      value ← cv.createValue();
      sum ← cv.getSum();
      sum ← (sum * workload)/100;
      avgPerNode ← sum/counted;
      value.setValue(avgPerNode);
      ressource.add(value);
    end for
    job.addRequirement(ressource);
    if counted = numberNodes then
      job ← newService();
      job.setAmount(NumberNodes);
      jobList.add(job);
    else
      job ← newConstraintJob();
      contraintList.add(job);
    end if
    job.setID(id);
    id = id + 1;
  end for
```

Der dafür notwendige Ablauf ist in Algorithmus 4 dargestellt. Alle ermittelten Ressourcen werden durchlaufen und eine Kopie erstellt, welche später für die Konfiguration benötigt wird. Im nächsten Schritt werden alle gesammelten Werte durchlaufen für die ebenfalls Kopien erzeugt werden.

Die Summe aller Werte wird an den `workload` angepasst und durch die Anzahl der Knoten, welche die Ressource haben geteilt. Das ergibt den durchschnittlichen Wert pro Knoten, welcher als Anforderung dienen wird und in der Ressource gespeichert wird.

Als nächstes wird per Fallunterscheidung geprüft, ob alle Knoten die Ressource besitzen. In diesem Fall kann ein Job erstellt werden, welcher als Anzahl genau die Knotenzahl bekommt. Damit existiert pro Knoten ein Job für diese Ressource mit den durchschnittlichen Werten. Für den anderen Fall wird mit einer Hilfsklasse ein Constraint erstellt.

Die generierte Konfiguration ist erfüllbar und verwendet eine feste Anzahl an Jobs. Ein Problem des Ergebnisses ist, dass es einen eher optimalen Fall beschreibt.

Bei den Testläufen wurde jeweils eine optimale Verteilung erreicht, da jeder Knoten einen Job einmal erbringt. Das Ergebnis ist eigentlich sehr schön, allerdings soll der Algorithmus seine Leistungsfähigkeit auch unter Beweis stellen mit einer Konfiguration, wie sie eher in der Praxis vorkommen wird.

6.1.2 Komplexer Generator

Die zweite Implementierung eines Generators sollte eine Konfiguration generieren, die stärker in der Anzahl der Jobs und benötigten Ressourcen variiert. Das Ergebnis sollte mehr dem Einsatz in der Praxis ähneln, wie es beispielsweise von einem Administrator für eine Applikation benutzt werden würde. Um eine etwas differenzierte Konfiguration zu ermöglichen, sind einiges mehr an Werten notwendig. Diese sind nicht festgelegt, sondern werden eingestellt, weshalb eine Konfigurationsdatei dafür notwendig ist. In Algorithmus 5 sind die notwendigen Vorbereitungen für die Generierung abgebildet.

Algorithmus 5 Komplexer Konfigurations Generator

Require: $nodes$, $numberNodes$, $workload$, $countingRessource$, $varyWorkload$, $maxProviders$, $multipleJobsPercentage$, $jobsForNode$

$jobList \leftarrow newList()$;
$id \leftarrow 1$;
$amountJobs \leftarrow (numberNodes * jobsForNode)/100$;
$amountMultipleJobs \leftarrow (amountJobs * multipleJobsPercentage)/100$;
$ressourceKeys \leftarrow contingRessource.getKeys().clone()$;

Neben den Werten für den einfachen Generator, werden noch zusätzlich folgende Werte benötigt:

- `varyWorkload`
  Ein boolscher Wert zur Steuerung, ob die erzeugten Anforderungen zufällig variiert werden sollen.

- `maxProviders`
  Die Obergrenze der Dienstleister, die ein Job haben darf.

- `multipleJobsPercentage`
  Prozentualer Anteil der erzeugten Dienste, die mehr als einen Dienstleister haben.

- `jobsForNode`
  In Abhängigkeit der Anzahl der Knoten werden Jobs erzeugt. Dieser Wert steuert das Verhältnis von Jobs pro Knoten und dient dem Generator als grober Richtwert.

Die generierten Jobs werden in einer Liste gemerkt und erhalten eindeutige Nummern. Um zu ermitteln, wie viele generiert werden sollen, wird der prozentuale Anteil von `jobsForNode` von der Anzahl `numberNodes` berechnet. Bei einem Wert von 100 für `jobsForNode` soll ein Job pro Knoten generiert werden.

Das ergibt den Wert `amountJobs` mit dem ermittelt werden kann, wie viel Prozent davon Jobs mit mehr als einem Dienstleister sind.

Als letzte Vorbereitung werden die Namen der vorhandenen Ressourcen ermittelt und eine Kopie erstellt. Die Anforderungen an einen Job werden im weiteren Verlauf zufällig ausgewählt. Das wird über die Reihenfolge der Einträge in der Liste bewerkstelligt, weshalb eine Kopie notwendig ist.

Damit kann der erste Teil der Generierung beginnen, wie in Algorithmus 6 in PseudoCode beschrieben.

Mit Hilfe des Wertes `amountJobs` wird eine Schleife durchlaufen, mit der ein Job erstellt werden soll. Zuerst werden die generiert, die mehr als einen Dienstleister haben. Der Grund dafür ist, dass diese in der Summe mehr Ressourcen benötigen und deshalb zuerst von den ermittelten Werte abgezogen werden sollten.

Die Anzahl der Dienstleister wird aus einem Intervall zufällig gewählt und ist durch die festgelegte Untergrenze von zwei und der wählbaren Obergrenze `maxProviders` beschränkt.

Im nächsten Schritt wird die Anzahl der Ressourcen zufällig ausgewählt, die als Anforderungen für den Dienst gesetzt werden sollen. Damit soll erreicht werden, dass sich die Jobs untereinander nicht zu stark ähneln und später im Ablauf die Knoten verschiedene Werte des Quality of Service berechnen. Um noch mehr Variation zu erreichen, werden die Namen der Ressourcen in eine zufällige Reihenfolge gebracht.

Das ermöglicht es, unter Benutzung einer Schleife die Ressourcen mittels ihres Namens auszuwählen. Für den Job wird eine Ressource benötigt, welche in die An-

Algorithmus 6 Komplexer Konfigurations Generator

```
for i ← 0; i < amountJobs; i ← i + 1 do
  job ← newService();
  job.setID(id);
  id ← id + 1;
  if amountMultipleJobs > 0 then
    amountMultipleJobs ← amountMultipleJobs − 1;
    providers ← getRandomValue(2, maxProviders);
    job.setAmount(providers);
  else
    job.setAmount(1);
  end if
  pickRessources ← getRandomValue(count(countingRessource));
  shuffle(ressourceKeys);
  for j ← 0; j < pickRessources; j ← j + 1 do
    cr ← countingRessource.get(ressourceKeys.get(j));
    ressource ← cr.createRessource();
    for CountingValue cv in cr do
      max ← (cv.getMaxValue() ∗ workload)/100;
      avgPerNode ← max/cr.getCounter();
      if varyWorkload then
        avgPerNode ← getRandomValue(avgPerNode);
      end if
      subtract ← cv.getSumValue() − (avgPerNode ∗ job.getAmount());
      if subtract ≥ 0 then
        cv.adjustSum(subtract);
        value ← cv.createValue();
        value.setValue(avgPerNode);
        ressource.add(value);
      end if
    end for
    if ressource.hasValues() then
      job.addRequirement(ressource);
    end if
  end for
  if job.hasRequirements() then
    joblist.add(job);
  end if
end for
```

forderungen aufgenommen werden kann. Für diesen Zweck wird eine Kopie aus der `CountingRessource` erzeugt.

Es müssen alle ermittelten `CountingValue` durchlaufen werden, ein maximaler Wert in Abhängigkeit der gewünschten Auslastung ermittelt und auf den durchschnittlichen Wert `avgPerNode` pro Knoten umgerechnet werden.

Eine weitere Variation kann über `varyWorkload` aktiviert werden, die `avgPerNode` durch einen zufälligen Wert zwischen 1 und `avgPerNode` ersetzt.

Jede `CountingValue` dient zugleich auch als Zähler der noch verbleibenden Kapazität für die gesetzte Auslastung[1]. Um den Verbrauch zu ermitteln, muss `avgPerNode` mit der Anzahl der Dienstleister multipliziert und von der Summe der noch vorhandenen abgezogen werden.

Ergibt sich hier ein negativer Wert so würde durch die Anforderung mehr verbraucht werden, als an Kapazität noch vorhanden ist. In dem Fall wird der Wert nicht hinzugefügt, damit die angestrebte Auslastung nicht verfälscht wird. Ansonsten wird ein Wert gebildet und der Ressource hinzugefügt.

Nachdem alle Werte durchlaufen sind, wird eine Ressource dem Job nur dann als Anforderungen hinzugefügt, wenn sie auch mindestens einen Wert hat.

Sind alle Ressourcen durchlaufen, wird die Anzahl der Anforderungen überprüft, die ein Job hat. Da ein Job ohne Anforderungen für die generierte Konfiguration uninteressant ist, werden diese verworfen.

Nach Ablauf der Schleife sind unter Umständen nicht genau `amountJobs` Jobs entstanden. Das ist auch in Ordnung, denn es soll nach Möglichkeit variiert werden. Allerdings ist zu diesem Zeitpunkt noch nicht sicher, dass auch genug Anforderungen erzeugt wurden, um die gewünschte Auslastung zu erreichen. Es muss nochmal über die ermittelten Ressourcen iteriert werden, um die verbliebene Kapazität zu ermitteln und in die Konfiguration einzubauen, wie in Algorithmus 7 beschrieben.

In der Menge `countingRessource` sind unter Umständen noch Restkapazitäten vorhanden, die als Anforderungen in jeweils einem Job verwendet werden sollen. Dafür wird eine Ressource (`mayBeAdded`) erstellt, die als vorläufige Anforderung die Summe der Restkapazitäten enthält.

Die Werte in der `CountingRessource` werden untersucht, um die eventuell verbliebene Kapazitäten zu ermitteln und diese in einem Job unterbringen. Ist noch Kapazität vorhanden, wird daraus eine erfüllbare Anforderung gestaltet. Dazu wird berechnet, wie viel durchschnittlich auf jeden Knoten fallen würde. Dieser Wert wird noch ein-

[1] Das ist im PseudoCode nicht dargestellt

Algorithmus 7 Komplexer Konfigurations Generator

```
for all CountingRessource cr in countingRessource do
  mayBeAdded ← cr.createRessource();
  amount ← 1;
  for all CountingValue cv in cr do
    if cv.getSumValue() > 0 then
      avgPerNode ← cv.getMaxValue()/numberNodes;
      adjusted ← avgPerNode/2;
      remaining ← cv.getSumValue();
      if remaining > adjusted then
        howoften ← remaining/avgPerNode;
        if howoften > numberNode then
          howoften ← numberNodes;
        end if
        jobValue ← remaining/howoften;
        amount ← howoften;
        value ← cv.createValue();
        value.setValue(jobValue);
        mayBeAdded.add(value);
      end if
    end if
  end for
  if mayBeAdded.hasValues() then
    job ← newService();
    job.addRequirement(mayBeAdded);
    job.setID(id);
    job.setAmount(amount);
    id ← id + 1;
    joblist.add(job);
  end if
end for
```

mal halbiert, da die Anforderungen an eine Ressource unter Umständen bereits schon zu hoch sind.

Sind die vorhandenen Ressourcen größer als dieser Wert, so lohnt sich eine Aufteilung auf mehrere Knoten. Dazu wird berechnet, auf wie viele Knoten sich die verbleibende Kapazität aufteilen lässt, wobei das Ergebnis natürlich die Anzahl der Knoten nicht überschreiten darf. Das bildet den `amount` des Jobs. Der Quotient aus der vorhandenen Kapazität und der Anzahl der Knoten ergibt den Wert, der im `Value` eingetragen wird. Das wird der Ressource hinzugefügt, aus der ein Job gebildet wird.

Damit ist die Generierung der Konfiguration abgeschlossen, die nur noch für den Simulator in eine XML Datei geschrieben werden muss.

6.2 Nachrichten zählen

Ein relevanter Wert für die Evaluation ist die Anzahl der Nachrichten, die für die Lösung der Aufgabe benötigt werden. Unicast und Broadcast Nachrichten werden unterschiedlich gezählt, dazu existiert in der entsprechenden Nachrichtenklasse ein Zähler wie in Abschnitt 3.3.1 beschrieben.

Jeder Zähler ist ein `AtomicInteger`, der nur mit unteilbaren Operationen verändert werden kann und sich somit sehr gut für den Einsatz in einem System mit sehr vielen Threads eignet.

Zusätzlich hat jeder Betreff einer Nachricht einen eigenen Zähler anhand dessen im Nachhinein ermittelt werden kann, welche Nachrichten verschickt wurden. Damit lässt sich ableiten, was während des Algorithmus passiert ist, also welche Phasen des Algorithmus durchlaufen wurden und welche Zustände eingetreten sind. Es lässt sich beispielsweise ermitteln, wie oft eine Konfiguration verteilt wurde und wie häufig gemeldet wurde, dass diese erfüllt sei. Weitere wichtige Zustände für die ein Zähler existiert, sind:

1. Ausführung oder Überschreibung eines Jobs durch einen Knoten.
2. Behebung von Inkonsistenzen, die bei der Verifikation der Konfiguration aufgetreten sind.
3. Anfrage an andere Knoten nach Informationen zu Diensten, die noch nicht erbracht sind.
4. Prüfung auf Isolation, wenn nach einer festgelegten Wartezeit keine Nachrichten mehr empfangen wurden.

Von besonderem Interesse für die Auswertung ist das Zählen von überschriebenen Jobs. Dabei können zwei unterschiedliche Fälle unterschieden werden, nämlich wenn zwei Knoten die Ausführung eines Jobs gleichzeitig bekannt geben und sobald ein Knoten entscheidet, dass er einen Job besser erbringen kann. Der erste Fall wird als Kollision bezeichnet und letzterer als Überschreibung.

Die Anzahl der Überschreibungen ist ein Wert für den es einen eigenen Zähler gibt, der hochgezählt wird, sobald im Algorithmus entschieden wird, einen Job zu überschreiben. Damit kann festgestellt werden, wie häufig es zu gewollten Überschreibungen kommt, die den Versand einer Nachricht zur Folge haben. Ein bestimmter Anteil der verschickten Nachrichten lässt sich damit den Überschreibungen zuordnen. In den Diagrammen findet sich ein Wert der als „ohne Überschreibung“ gekennzeichnet ist, der angibt, wie viele Nachrichten benötigt worden wären, wenn kein Knoten einen Job überschrieben hätte.

6.3 Auswertung der Testläufe

Für die Auswertung wurde die Simulation im Batchmodus gestartet. Der Algorithmus wurde mit verschiedenen Parameter in jeweils 100 Simulationsläufen getestet, um eine Bewertungsgrundlage für das Verhalten und die Güte des Algorithmus zu erhalten.

Homogene Hardware
In Sensornetzwerken oder im Gridcomputing werden viele Rechner vernetzt, die häufig die gleiche Hardwarekonfiguration aufweisen. In der Simulation hat jeder Knoten exakt die gleiche Hardware, um diesen Fall abzubilden.

Heterogene Hardware
Es soll ein Netzwerk simuliert werden, in dem sich die Rechner in ihrer Art und ihrer Leistungsfähigkeit unterscheiden. Beispielsweise würden neben Servern auch PDAs an der Aushandlung teilnehmen. Die Werte der Ressourcen der Knoten werden in definierten Grenzen variiert, aber nicht die Anzahl der Ressourcen selbst, da dies selbst einen getesteten Parameter darstellt.

Jeder Testlauf wurde einmal mit homogener und einmal mit heterogener Hardware durchgeführt.

Anzahl Ressourcen
Jeder Knoten hat eine definierte Anzahl an Ressourcen in seiner Hardware. Eine höhere Anzahl an Ressourcen bedeutet mehr Variation in der Konfiguration. Je mehr Ressourcen zur Verfügung stehen, desto eher werden Jobs generiert, die unterschiedliche Ressourcen benötigen. Es wurde immer mit 3, 5, 10 und 15 Ressourcen getestet.

Auslastung
Die Auslastung gibt an, wie viel Prozent der Gesamtkapazität der Knoten für eine Konfiguration verwendet werden soll. Dieser Wert wird in der generierten Konfiguration verwendet, um die Werte der Anforderungen zu gestalten. Im Schnitt sollte ebenfalls die gleiche Auslastung erreicht werden, was auf den einzelnen Knoten stark davon abhängt, wie die Hardware der Knoten beschaffen ist und wie gut sich die Konfiguration verteilen lässt. Die Auslastung wurde mit den Werten 20%, 60%, 80% und 100% getestet.

Knotenzahl
Mit der Anzahl der Knoten, die an der Simulation teilnehmen, kann das Verhalten bei großen Netzwerken getestet werden. Das ist gerade deswegen von Interesse, da die Aufgabe der Verteilung in großen Netzwerken eine sehr große Anzahl an Nachrichten benötigt. Die generierte Konfiguration sorgt für eine entsprechende Anzahl Jobs, die der Anzahl der Knoten angepasst ist. Es wurde mit 10, 25, 50 und 100 Knoten getestet.

6.3.1 Simulationsläufe

Als ein direktes Ergebnis der Simulationsläufe werden die verwendeten Broadcastnachrichten betrachtet. Unicastnachrichten werden nur im Fehlerfall benötigt, die in den nachfolgenden Evaluationen nicht betrachtet wurden.

Es werden zwei Fälle definiert, mit deren Hilfe eine Einschätzung des Ergebnisses möglich ist. Dafür wird die Anzahl der Knoten n und die Gesamtzahl der Dienstleister m benötigt. Letztere Zahl ergibt sich durch Aufsummieren der Anzahl der Dienstleister, die in jedem Job spezifiziert ist. Um die Konfiguration zu erfüllen, wird für jeden Dienstleister mindestens eine Nachricht benötigt. Es wird zusätzlich noch eine Nachricht benötigt, um die Konfiguration zu verteilen und mindestens eine mit der gemeldet wird, dass sie erfüllt ist. Damit ergibt sich der *optimal case* mit $2 + m$. Häufig ergibt es sich, dass mehrere Knoten gleichzeitig fertig werden und damit im schlimmsten Fall jeder eine Nachricht schickt, dass die Konfiguration erfüllt ist. Dieser Fall wird als *suboptimal case* bezeichnet und berechnet sich als $1 + n + m$.

Auf ein richtiges Worst Case Szenario wurde verzichtet, da es unklar ist, wo dies genau anzusetzen ist, um eine Aussagekraft zu erhalten.

6.3.1.1 Heterogene Hardware

Zuerst werden die Ergebnisse bei heterogener Hardware und 80% Auslastung betrachtet, wobei 5 Ressourcen verwendet wurden. Dazu werden die Simulationsläufe zu 25,

50 und 100 Knoten in den Abbildungen 6.1, 6.2 und 6.3 dargestellt. Der Simulationslauf mit 10 Knoten bringt keine neuen Erkenntnisse und ist aus Platzgründen nicht abgebildet.

Zuerst fällt auf, dass die Ergebnisse stark schwanken, was damit zu erklären ist, dass die Konfigurationen stark variieren. Das hat zur Folge, dass die Zahl der zu erbringenden Jobs jeweils unterschiedlich hoch ist. Werden die Kurven der Nachrichten und die ohne Überschreibungen betrachtet, so folgen diese in den meisten Fällen den Schwankungen.

Das dem nicht immer so ist, kann an einem Fall in Diagramm 6.1 gesehen werden. Bei Simulationslauf 59 werden deutlich mehr Nachrichten benötigt. Wird der Wert ohne Überschreibungen zusätzlich betrachtet, so kann dies damit erklärt werden, dass es in diesem Durchlauf zu vielen Kollisionen gekommen ist. Viele Knoten wollten gleichzeitig einen Job übernehmen und haben damit mehr Nachrichten verursacht.

Während sich bei 25 Knoten die Kurven der Nachrichten noch eng am *suboptimal case* bewegt haben, so zeichnet sich in Diagramm 6.2 bereits ab, dass dies nicht so bleibt. Es werden deutlich mehr Nachrichten benötigt, wie in Abbildung 6.3 deutlich zu sehen ist.

Wird die Anzahl der notwendigen Nachrichten ohne Überschreibungen betrachtet, so fällt auf, dass sich diese immer unter dem *suboptimal case* befinden. Selbst wenn es mehr Knoten gibt, steigt die Kurve nicht über den Wert des *suboptimal case*. Bei großen Netzwerken werden auch meist weniger Nachrichten benötigt, da bei vielen Knoten die Nachricht, dass die Konfiguration erfüllt ist, häufig erhalten wurde bevor selbst eine geschickt wird.

6.3.1.2 Homogene Hardware

Werden die Diagramme der heterogenen Hardware in den Abbildungen 6.4, 6.5 und 6.6 zum Vergleich herangezogen, so ähneln sich diese bei wenigen Knoten stark. Beide teilen die starken Schwankungen. Nur verläuft die Kurve der verschickten Nachrichten sehr nah an und teilweise unter der des *suboptimal case*, während bei heterogener Hardware die Kurve bereits darüber war.

Werden die Diagramme 6.5 und 6.6 betrachtet, so hebt sich die Kurve nicht stark vom *suboptimal case* ab. Der Algorithmus funktioniert besser bei homogener Hardware, wie ersichtlich wird, wenn die Diagramme der heterogenen Hardware als Vergleich herangezogen werden. Das ist beispielsweise besonders gut im Vergleich mit Abbildung 6.3 zu erkennen.

Werden die Information der verschickten und der benötigten Nachrichten betrachtet, so ergibt sich der Verdacht, dass dies auf die hohe Anzahl an Überschreibungen

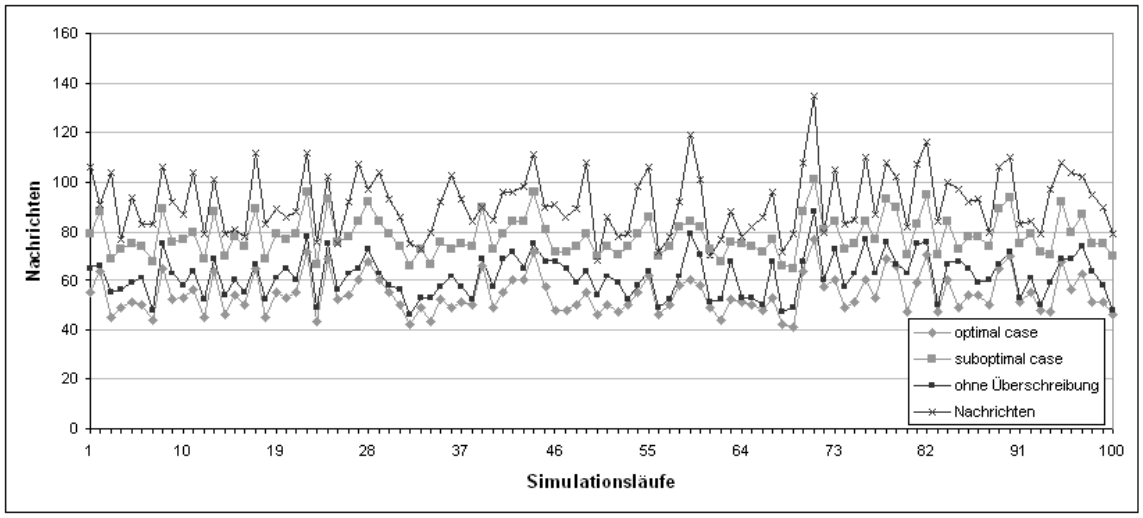


Abbildung 6.1: 25 Knoten mit heterogener Hardware

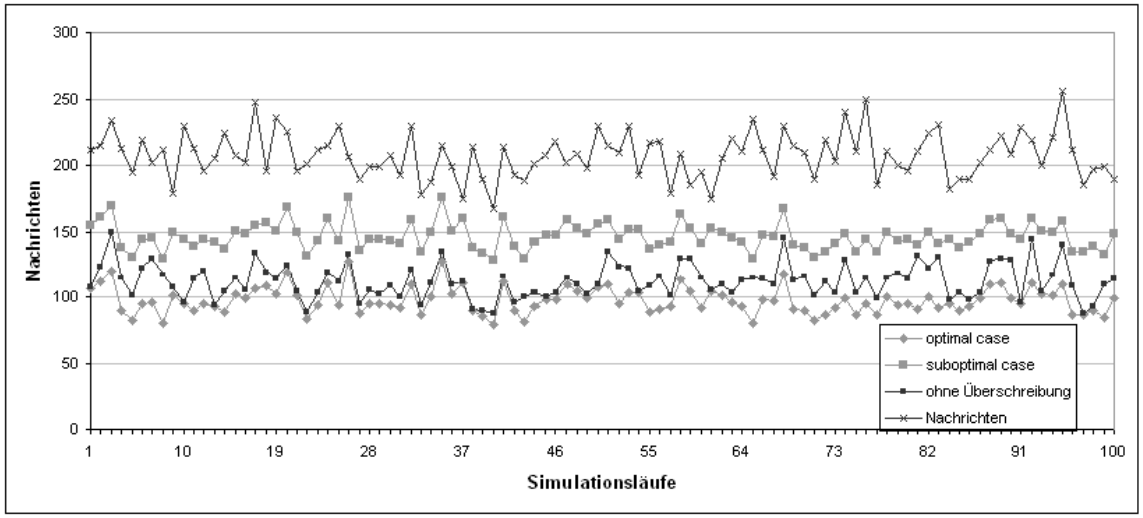


Abbildung 6.2: 50 Knoten mit heterogener Hardware

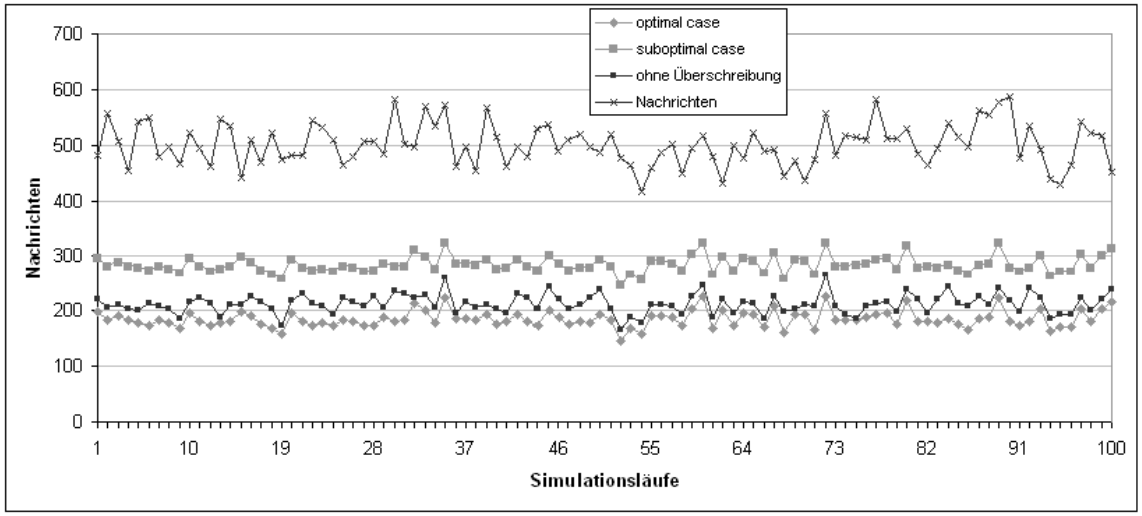


Abbildung 6.3: 100 Knoten mit heterogener Hardware

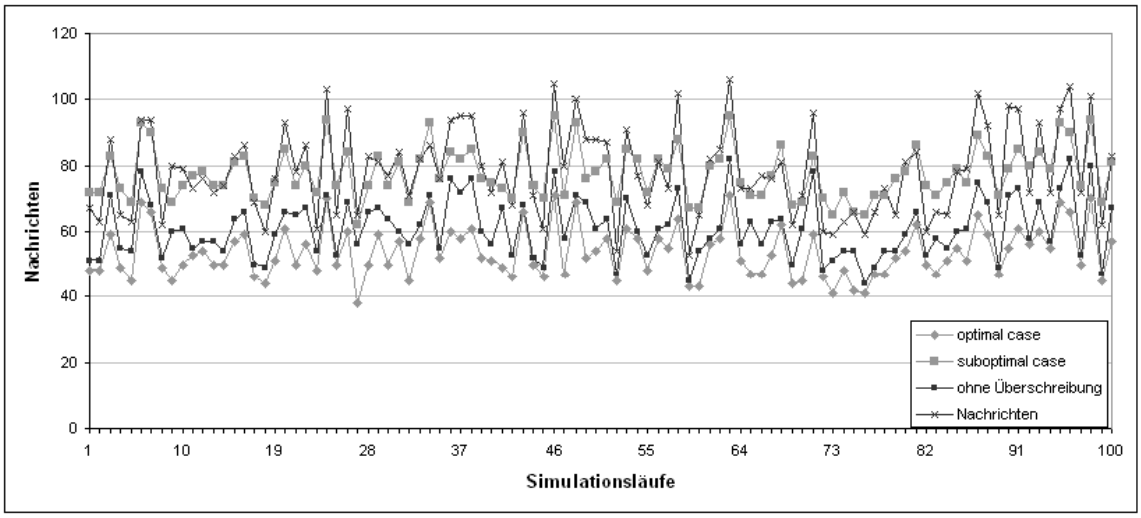


Abbildung 6.4: 25 Knoten mit homogener Hardware

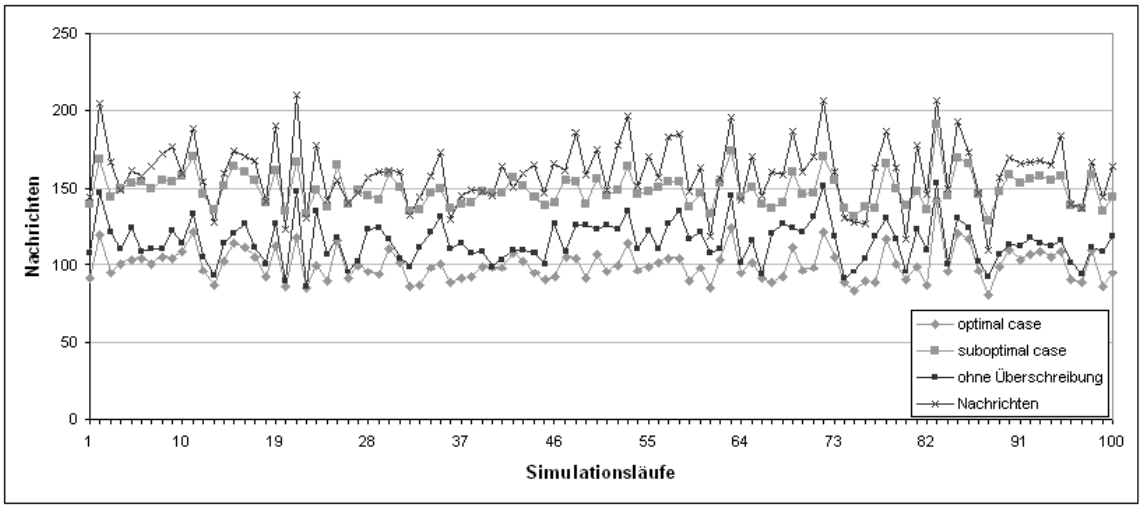


Abbildung 6.5: 50 Knoten mit homogener Hardware

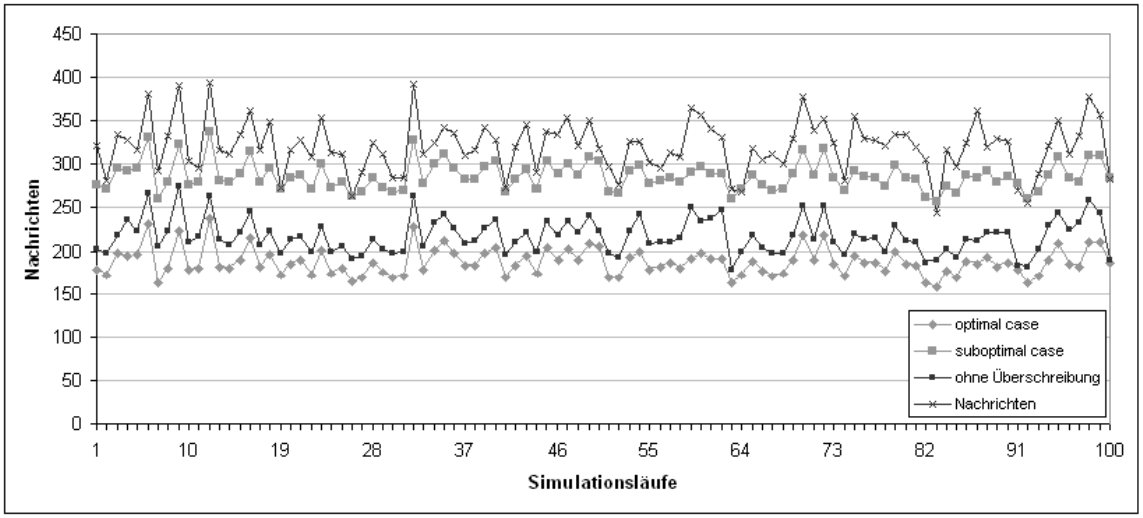


Abbildung 6.6: 100 Knoten mit homogener Hardware

zurückzuführen ist. Dazu muss aufgeschlüsselt werden, welche Arten von Nachrichten verschickt wurden, um damit Rückschlüsse auf den Ablauf ziehen zu können.

In der Abbildung 6.7 wurden die verschickten Nachrichten der Datenreihen aufgeschlüsselt, die den Diagrammen 6.3 und 6.6 zu Grunde liegen. Zu Beachten ist, dass die Werte alle nur die durchschnittliche Anzahl darstellen und die Gesamtanzahl an Nachrichten bei homogener Hardware geringer ist. Trotzdem zeigt sich, dass ein Großteil der verschickten Nachrichten bei heterogener Hardware auf Überschreibungen zurückzuführen ist, was dazu führt, dass viel mehr kommuniziert wurde als bei homogener Hardware.

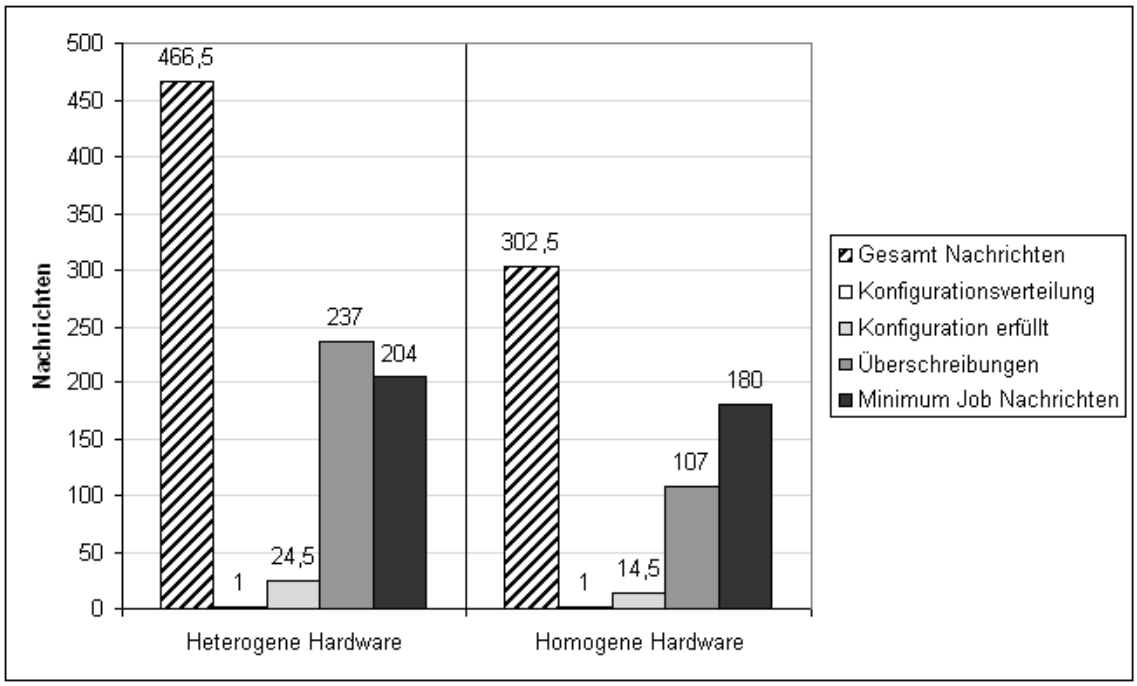


Abbildung 6.7: Vergleich der Nachrichten

Es stellt sich die Frage, warum es gerade bei homogener Hardware zu weniger Überschreibungen kommt. Bei einem Durchlauf des Algorithmus wird ein Job mit der Metrik bewertet. Da jeder Knoten die gleiche Hardware aufweist entsteht die gleiche Bewertungsgrundlage und häufig wird keine Verbesserung eintreten, weshalb es nicht zu einer Überschreibung kommt.

Anders verhält es sich bei heterogener Hardware. Dort wird jeder Job zwangsläufig unterschiedlich bewertet, auch wenn es sich nur um wenige Punkte handelt. Damit gibt es stärkere und schwächere Knoten. Es ist nicht garantiert, dass immer die „guten“ Knoten einen Job ausführen wollen, sondern es kann durchaus ein „schlechter“ sein. Kommen diese später an die Reihe, so werden diese eine bessere Quality of

Service berechnen und den Job selbst erbringen. Der schlimmste Fall wäre, wenn der schlechteste Knoten beginnt und der jeweils nächste nur ein wenig besser ist, was eine Kaskade an Nachrichten nach sich ziehen würde.

Eine Überschreibung lohnt sich nur, wenn wirklich etwas verbessert wird. Dies ist bei einem Unterschied von einem oder zwei Punkten nicht der Fall. Es bedarf also einer Schwelle ab der überschrieben werden darf. Beispielsweise könnte dies der Fall sein, wenn sich der Quality of Service um mindestens 10% verbessert. Dazu sollte aber eine Abwägung zwischen Kommunikationskosten und der eigentlichen Verbesserung erfolgen und der Wert sollte durch Simulation ermittelt werden.

Erfolgversprechender ist es zu verhinden, dass die schlechten Knoten zuerst an die Reihe kommen um einen Job auszuführen. Damit könnte die Anzahl der Nachrichten entscheidend verringert werden. Aber woher weiß ein Knoten, dass er eine schlechtere Quality of Service hat? Da nichts über die anderen Knoten bekannt ist, muss diese Informationen durch Analyse des Nachrichtenverkehrs gewonnen werden. Anhand des Quality of Service anderer Knoten kann ermittelt werden, ob ein Job gut ausgeführt werden kann. Damit kann das Versenden der Nachricht verzögert werden, um besseren Knoten die Chance zu geben, zuerst den Job zu belegen.

6.3.1.3 Variation der Ressourcen

Wie bereits erwähnt, gab es auch weitere Testläufe mit unterschiedlicher Auslastung und auch verschiedenen Ressourcen. In der Abbildung 6.8 zeigt eine Übersicht der Abläufe bei unterschiedlicher Knoten- und Ressourcenzahl. Beachtenswert sind die Ergebnisse bei homogener Hardware mit 50 und 100 Knoten. Das Ergebnis verschlechtert sich zunehmend, je mehr Ressourcen verwendet werden.

Es liegt die Vermutung nahe, dass es sich hierbei um einen ähnlichen Effekt handelt, wie im heterogenen Fall. Die Ursache könnte in der Konfiguration zu finden sein, denn je mehr Ressourcen vorhanden sind, desto unterschiedlicher gestalten sich die Anforderungen der generierten Jobs. Damit werden verschiedene Ressourcen belegt und damit verhält es sich ähnlich, als ob jeder Knoten unterschiedliche Hardware hätte. Damit sollten die oben erwähnten Verbesserung auch hier Wirkung zeigen.

6.3.1.4 Variation der Auslastung

Als nächstes wird das Verhalten der Simulation bei unterschiedlicher Auslastung untersucht. In jeweils 100 Durchläufen wurden für alle Knoten jeweils mit 20%, 60%, 80% und 100% Auslastung getestet. Es gibt kaum Auswirkungen auf die Anzahl der Nachrichten überraschenderweise mit Ausnahme der homogenen Hardware. Bei 100%

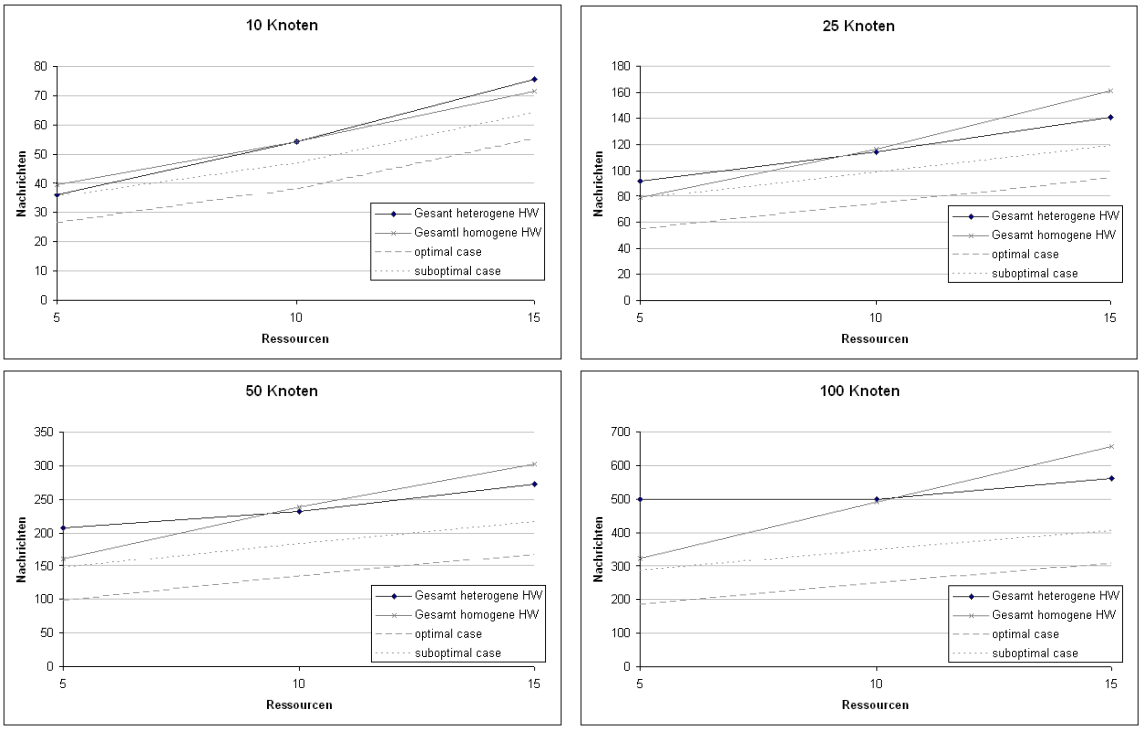


Abbildung 6.8: Nachrichtenanzahl bei verschiedenen Ressourcen

Auslastung werden weniger Nachrichten benötigt, als bei geringerer Auslastung, wie in Abbildung 6.9 zu sehen ist.

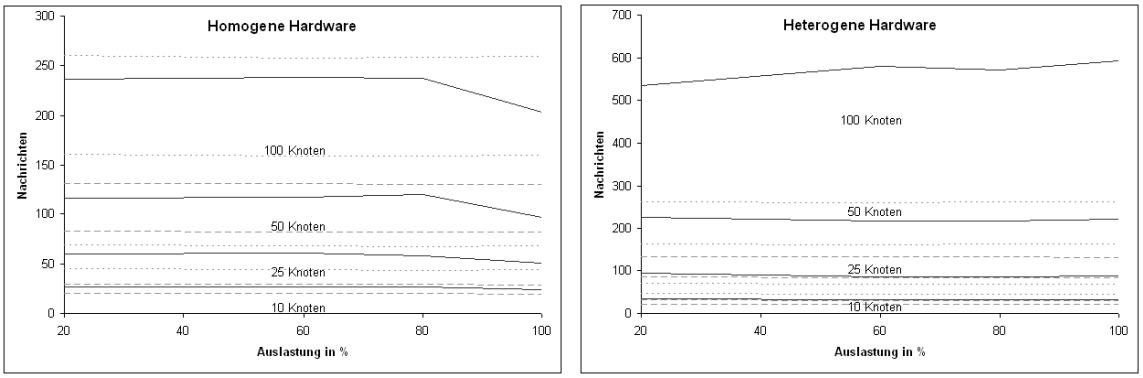


Abbildung 6.9: Nachrichten pro Auslastung

Das Ergebnis wird deutlicher, wenn die durchschnittliche Anzahl an Nachrichten pro Job betrachtet wird, die in Abbildung 6.10 dargestellt ist. Bei maximaler Auslastung werden in der Tat weniger Nachrichten benötigt, wobei sich noch kleine Unterschiede bei der Knotenanzahl zeigen. Je mehr Knoten desto weniger Nachrichten werden benötigt. Bei heterogener Hardware stellt sich das Bild anders dar. Dort werden

mehr Nachrichten benötigt, je mehr Knoten teilnehmen. Wie aber bereits beschrieben, hängt das zum Teil mit der generell schlechteren Leistung im heterogenen Fall zusammen.

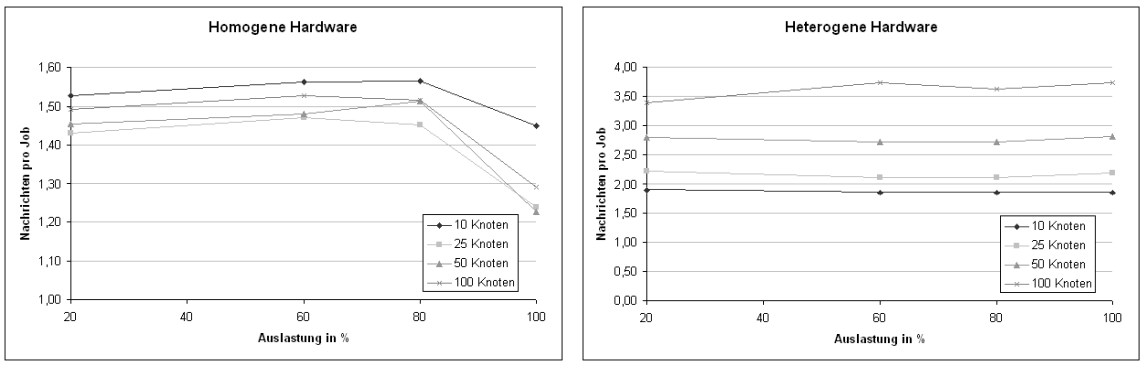


Abbildung 6.10: Durchschnittliche Anzahl Nachrichten pro Auslastung

6.3.1.5 Anzahl der Nachrichten

Mit deterministischen Algorithmen steigt die Anzahl der Nachrichten die für eine Verteilung der Dienste benötigt werden, exponentiell mit der Größe des Netzwerks an. Die Anzahl der Jobs beeinflusst, wie viele Nachrichten verschickt werden. Ziel dieser Arbeit war es auch, das Verhältnis von benötigen Nachrichten pro Job möglichst gering zu halten, es sollte also nicht exponentiell steigen.

In Abbildung 6.11 wird eine Übersicht der Simulationsläufe dargestellt. Die verschiedenen Knotenanzahlen wurden in einem Diagramm dargestellt, um die Ergebnisse zu vergleichen. Auf der x-Achse befindet sich die Knotenanzahl und auf der y-Achse das Verhältnis Nachrichten pro Job. Im linkem Diagramm ist jeweils die homogene und im rechten die heterogene Hardware abgebildet. Die erste Reihe zeigt die Läufe mit 3, gefolgt von 5, 10 und zuletzt mit 15 Ressourcen.

Der Unterschied zwischen homogener und heterogener Hardware ist bei drei Ressourcen am größten. Es scheint am Anfang so, als wäre der Verlauf bei homogener Hardware konstant. Das heißt unabhängig von der Anzahl der Knoten würden gleich viel Nachrichten pro Job benötigt. Werden aber die Diagramme mit mehr Ressourcen betrachtet, so werden dennoch mehr Nachrichten pro Knoten benötigt. Bei heterogener Hardware scheint sich bei drei Ressourcen ein schlechteres Ergebnis abzuzeichnen, was sich aber mit steigender Ressourcenzahl zunehmend bessert. Je mehr Ressourcen verwendet werden desto mehr gleichen sich die Kurven von heterogener und homogener Hardware. Es werden zwar nicht konstant viele Nachrichten benötigt, aber immerhin wächst die Funktion annähernd linear.

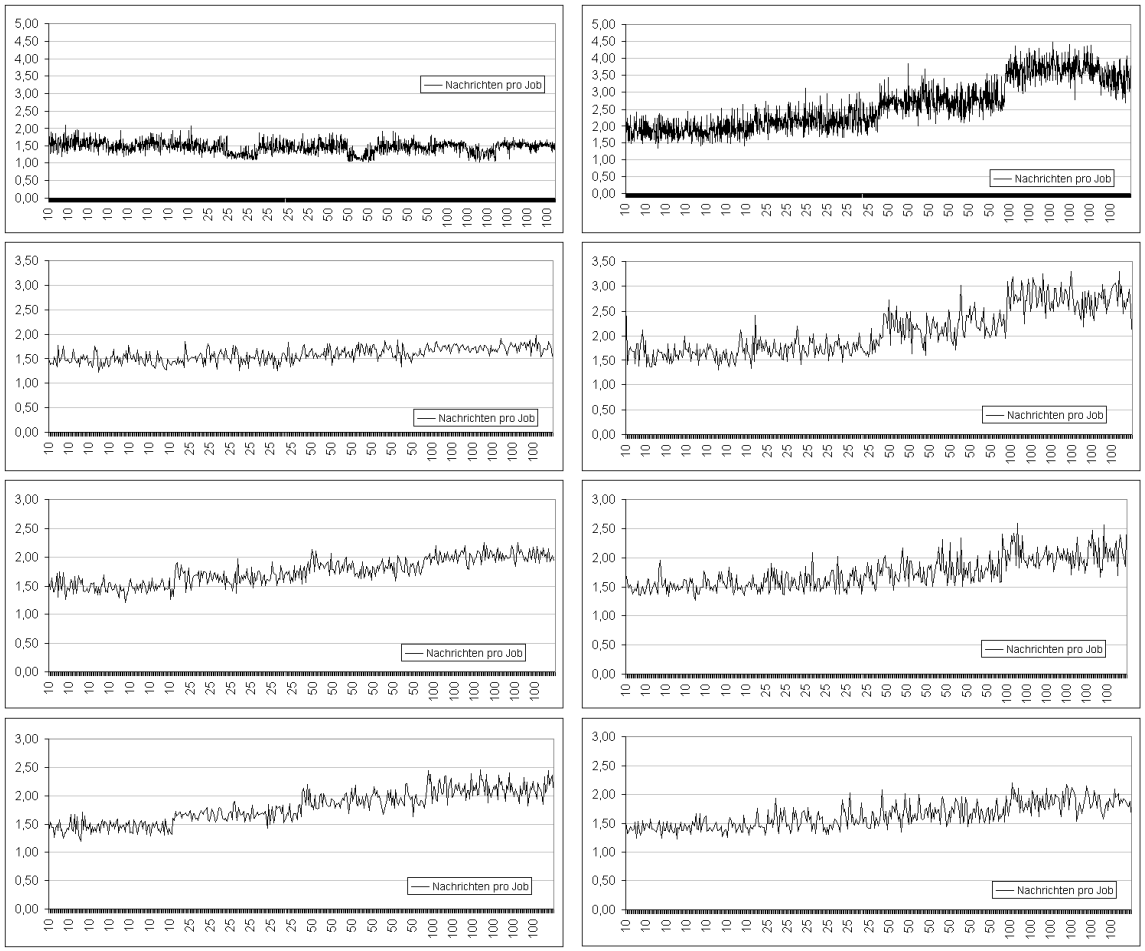


Abbildung 6.11: Nachrichten pro Job

6.3.1.6 Erste Optimierung

In der ersten Implementierung wurde eine initiale Bewertung der Jobs verwendet, um pro Durchlauf zu entscheiden, welcher Job als bester erbracht werden kann. Durch eine fortschreitende Belegung können aber Ressourcen unterschiedlich stark ausgelastet werden, was das Ergebnis der initialen Bewertung verfälscht. Eine einfache Optimierung besteht darin, bei jedem Durchlauf zuerst alle noch offenen Jobs neu zu bewerten und zu sortieren, anstatt die initiale Bewertung zu verwenden.

Die optimierte Variante wurde mit 5 Ressourcen und 80% Auslastung mit verschiedener Knotenanzahl getestet und mit dem alten Ergebnis verglichen. Das Verfahren senkt die Überschreibungen und damit die verschickten Nachrichten bei homogener Hardware, wie in Abbildung 6.12 zu sehen ist. Damit sich die Ergebnisse vergleichen lassen, zeigen die Kurven die benötigten Nachrichten im Verhältnis zum optimalen Wert. Leider stellt sich dieser Erfolg nicht bei heterogener Hardware ein, da es dort

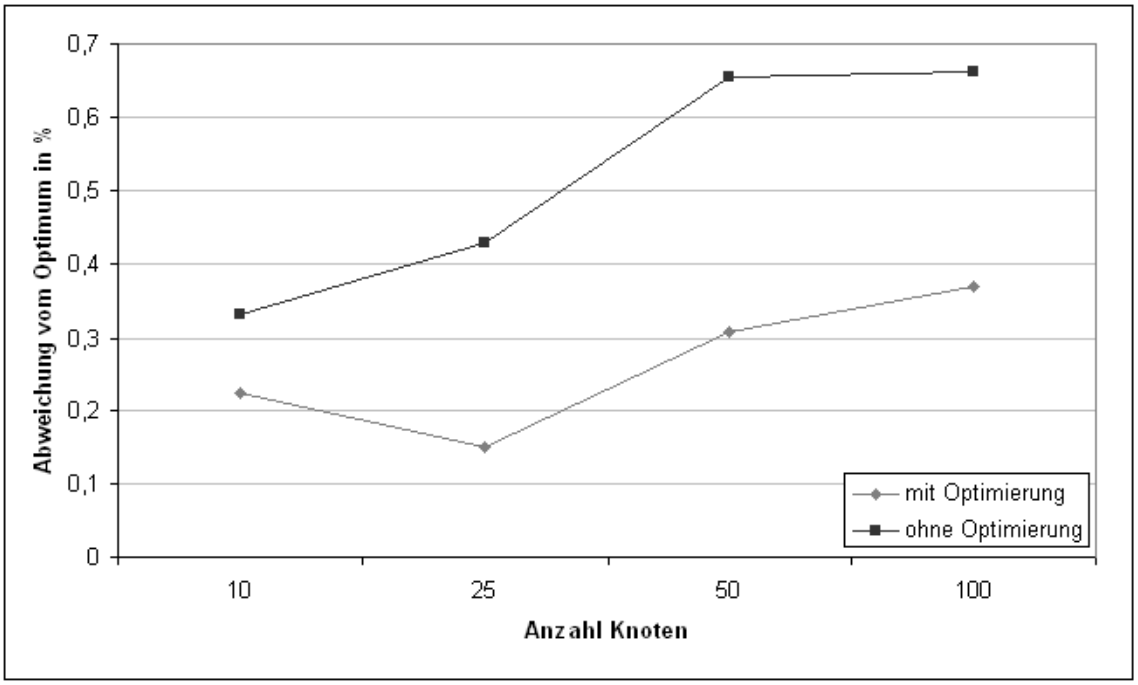


Abbildung 6.12: Verbesserungen durch Optimierung

zu mehr Überschreibungen führt, was den Effekt wieder aufhebt.

6.3.1.7 Fazit

Die Auswertung ergab einige positive Ergebnis, zeigt jedoch auch die Probleme, die noch gelöst werden sollten. Wie schon erwähnt, muss die Anzahl der Überschreibungen gesenkt werden. Die Verteilung der Dienste muss genauer untersucht werden, da die ermittelten Werte zu ungenau sind, um Rückschlüsse auf die Leistungsfähigkeit des Algorithmus zu ziehen. Dazu wäre es erforderlich eine Möglichkeit zu finden, wie gut sich eine Konfiguration beziehungsweise die einzelnen Jobs verteilen lassen.

Die Simulation wurde auch mit künstlichem Nachrichtenverlust getestet, jedoch waren die Ergebnisse unbefriedigend. Treten in der Kommunikation wenige Fehler auf, so kommt das System damit gut zurecht. Jedoch kann es passieren, dass durch den Verlust mehrerer Nachrichten die Fehlerkorrektur selbst zu einer Unmenge an Nachrichten führt. Wird bei der Verifizierung einer Konfiguration Fehler gefunden, so werden Verbesserungen per Broadcast verschickt. Diese Prozedur wird von jedem Knoten durchgeführt und eine Reaktion erfolgt nur, sofern die Verbesserung nicht schon erhalten worden ist. In den durchgeführten Testläufen mit Fehlerfall entstand eine hohe Anzahl an Nachrichten, die nicht akzeptabel ist. Verschärfend kam hinzu, dass sich der Nachrichtenverlust auf verschiedene Jobs ausgewirkt hat. Damit hatten

vielen Knoten falsche Werte an unterschiedlichen Stellen ihrer Konfiguration, was natürlich eine erhöhte Fehlerkorrektur bewirkte.

Auch für diesen Fall gibt es einen Lösungsvorschlag. Bei jeder Nachricht in der ein Knoten bekannt gibt, dass er einen Job erbringt, werden die vorher empfangenen Nachrichten hinzugefügt. Da die Information der Dienstleister redundant im Netzwerk sind, kann eine Korrektur während der Aushandlung durchgeführt werden, ohne zusätzliche Kommunikation. Der Preis dafür ist die Größe der Nachrichten, die noch die zusätzlichen Informationen aufnehmen muss, die aber nicht zu groß sind.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Ziel dieser Arbeit war die Entwicklung eines kooperativen Algorithmus für die Selbstkonfiguration der AMUN Middleware. Es wurde ein Modell geschaffen, das die gleichen Bedingungen wie AMUN für einen Simulator bietet. Die Abbildung umfasste neben den Nachrichten auch die Infrastruktur. Für die Selbstkonfiguration war es notwendig, eine Konfigurationsbeschreibung zu entwickeln, die Dienste und Ressourcen beschreibt, leicht erweiterbar und gut zu verarbeiten ist. Es wurde eine Metrik entwickelt, die einem Knoten die Bewertung der Güte eines Dienstes ermöglicht.

Mit diesen Voraussetzungen konnte ein Algorithmus entwickelt werden, der ein soziales Verhalten nachbildet. Die Knoten des Netzwerks verhandeln über die Ausführung der Jobs, die zu einem konsistentem Ergebnis der Belegung führt. Ermöglicht wurde dies durch spezielle Verfahren zur Konfliktvermeidung der Nachrichten. Mit dem Simulator, der im Rahmen dieser Diplomarbeit entworfen wurde konnte der Algorithmus unter verschiedenen Bedingungen getestet und die Ergebnisse ausgewertet werden.

7.2 Ausblick

Die Lösung der Aushandlung mit einem kooperativen Algorithmus, zeigt schon recht gute Ergebnisse, obwohl es noch einige Dinge zu verbessern gibt. In der Simulation hat sich gezeigt, dass die momentane Lösung zur Fehlerkorrektur von Inkonsistenzen bei der Verifikation nicht optimal ist. Ebenso verhält es sich mit der Anzahl der Überschreibungen, die im heterogenen Fall recht hoch ist. Dazu wurden bereits Lösungsvorschläge beschrieben.

Mit Hilfe der Analyse des Nachrichtenverkehrs könnten Knoten selbst ermitteln, wie gut ihre eigene Dienstgüte im Verhältnis zur letzten Belegung. Damit kann eine Wartezeit errechnet werden, so dass Knoten mit einen Dienst mit schlechter Dienstgüte möglichst spät belegen, was zu weniger Überschreibungen führen kann.

Neben der bereits betrachteten Optimierung bietet die Anwendung einer zusätzlichen Metrik zur Sortierung der Jobliste einen guten Ansatzpunkt für weitere Optimierungen, da sich mit der Reihenfolge entscheidet, welcher Job als nächstes von einem Knoten zur Ausführung markiert wird.

Die AMUN Middleware wird am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme der Universität Augsburg kontinuierlich weiterentwickelt. Der nächste Schritt besteht darin, die Selbstkonfiguration einzubauen, die auf den Ergebnisse dieser Arbeit basiert. Dort wird sie unter realen Bedingungen getestet und evaluiert. In Zukunft könnte dann begonnen werden, Applikationen zu entwickeln, die das Verfahren nutzen.

A Einstellungen für den Batchmodus

Ein Simulationslauf im Batchbetrieb kann für verschiedene Möglichkeiten konfiguriert werden. Die dazu notwendigen Einstellungen werden in einer Datei vorgenommen, wie etwa im folgenden Beispiel.

```
<?xml version="1.0" encoding="UTF-8"?>
<system-conf vary-hardware="false"
    amount-nodes-to-generate="25" generate-configuration="true">
  <default-hardware-conf>
    file:/D:/xml/DefaultHardWare.xml
  </default-hardware-conf>

  <config-schema>file:/D:/xml/configuration.xsd</config-schema>
  <mapping-file>file:/D:/xml/createdmappings.xml</mapping-file>

  <configuration-generator>Generator.xml</configuration-generator>

  <config-file>file:/D:/xml/configuration.xml</config-file>
  <network-conf></network-conf>
  <injector-conf></injector-conf>
  <base-hardware-conf>file:/D:/xml/5Ressorces.xml</base-hardware-conf>
</system-conf>
```

Viele der Einstellungen beziehen sich auf die Generierung der Knoten und des Netzwerks, obwohl auch vorgesehen ist, diese per Datei vorzugeben.

Die Simulatorkonfiguration wird durch das Element `system-conf` eingeleitet. Für die Generierung wird durch das Attribut `amount-nodes-to-generate` die Anzahl der Knoten angegeben und mit dem Attribut `vary-hardware` kann eingestellt werden, ob homogene oder heterogene Hardware für das Netzwerk verwendet werden soll.

Die Basis dafür ist in der Datei zu finden, welche im Element `base-hardware-conf` angegeben wird. Ähnlich wie in der Oberfläche wird eine Schablone definiert mit der die Hardware der einzelnen Knoten erzeugt wird.

Soll eine Konfiguration generiert werden, so muss `generate-configuration` auf „true“ gesetzt werden und das Element `configuration-generator` muss den Pfad einer Konfigurationsdatei für einen Konfigurationsgenerator enthalten. Ansonsten muss im Element `config-file` eine gültige Konfiguration angegeben sein oder die Simulation wird mit einer Fehlermeldung abgebrochen.

Sollen Fehler in die Kommunikation eingeschleust werden, so werden die dafür notwendigen Werte in der Datei im Element `injector-conf` eingetragen.

Alternativ zur Generierung war es vorgesehen, dass ein vordefiniertes Netzwerk verwendet werden kann, dass im Element `network-conf` eingetragen wird.

Für die korrekte Funktion des Simulators, sind noch zwei Einträge notwendig, einmal die Angabe des XML-Schemas zur Validierung der Konfiguration im Element `config-schema` und die Angabe der Mapping Regeln im Element `mapping-file`.

Literaturverzeichnis

[BHG+03] Blumenthal, Jan, Matthias Handy, Frank Golatowski, Marc Haase und Dirk Timmermann: *Wireless Sensor Networks - New Challenges in Software Engineering.* In: *9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Lissabon, Portugal, September 2003.

[BJH+04] Belecheanu, Roxana, Gawesh Jawaheer, Asher Hoskins, Julie A. McCann und Terry Payne: *Semantic Web Meets Autonomic Ubicomp.* In: *ISWC'04 Workshop on Semantic Web Technology for Mobile and Ubiquitous Applications*, Hiroshima, Japan, November 2004.

[BOS01] Boutaba, R., S. Omari und A. Singh: *SELFCON: An Architecture for Self-Configuration of Networks.* International Journal of Communications and Networks (special issue on Management of New Networking Infrastructure and Services), 3(4):317–323, 2001.

[cas05] *The Castor Project.* http://www.castor.org/index.html, September 2005.

[CGG+05] Curino, Carlo, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy und Gian Pietro Picco: TinyLime*: Bridging Mobile and Sensor Networks through Middleware.* In: *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, Seiten 61–72, Kauai Island (Hawaii, USA), März 2005. IEEE Computer Society.

[Con03] Constantinescu, Zoran: *Towards an Autonomic Distributed Computing System.* In: *DEXA Workshops*, Seiten 694–698, 2003.

[DHX+03] Dong, X., S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri und S. Rao: *Autonomia: An Autonomic Computing Environment.* In: *International Performance Computing and Communications Conference (IPCCC)*, Seiten 61–68, Phoenix, AZ, USA, April 2003.

[FRL05] Fok, Chien-Liang, Gruia-Catalin Roman und Chenyang Lu: *Mobile Agent Middleware for Sensor Networks: An Application Case Study.* In: *International Conference on Information Processing in Sensor Networks (IPSN'05), Special track on Platform, Tools, and Design Methods for Network Embedded Sensors (SPOTS)*, Los Angeles, CA, April 2005.

[HCL05] Heer, Jeffrey, Stuart K. Card und James A. Landay: *prefuse: a toolkit for interactive information visualization.* In: *CHI '05: Proceeding of the SIGCHI conference on Human factors in computing systems*, Seiten 421–430, New York, NY, USA, 2005. ACM Press.

[KC03] Kephart, Jeffrey O. und David M. Chess: *The Vision of Autonomic Computing.* Computer, 36(1):41–50, 2003.

[KMF90] Kramer, J., J. Magee und A. Finkelstein: *A Constructive Approach to the Design of Distributed Systems.* In: *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, Seiten 580–587, Washington, DC, 1990. IEEE Computer Society.

[KWBF00] Krishnamachari, B., S. Wicker, R. Bejar und C. Fernandez: *On the Complexity of Distributed Self-Configuration in Wireless Networks.* Telecommunication Systems, Special Issue on Wireless Networks and Mobile Computing, Seiten 169–177, 2000.

[LM03] Liu, T. und M. Martonosi: *Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems.* ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, June 2003.

[log05] *Logging Services Project.* http://logging.apache.org/, 2005.

[Man05] Mannhaupt, Danko: *Log4j JDBCAppender.* http://www.mannhaupt.com/danko/projects/, 2005.

[MDK92] Magee, Jeff, Naranker Dulay und Jeffrey Kramer: *Structuring Parallel and Distributed Programs.* In: *Proceedings of the International Workshop on Configurable Distributed Systems*, London, 1992.

[NM00] Neugebauer, R. und D. McAuley: *Congestion prices as feedback signals: An approach to QoS management.* In: *9th ACM SIGOPS European Workshop*, Seiten 91–96, Kolding, Denmark, September 2000.

[Obj05a] Object Management Group: *UML OCL 2.0 Specification.* http://www.omg.org/technology/documents/modeling_spec_catalog.htm, June 2005.

[Obj05b] Object Management Group: *UML Resource Page*. http://www.uml.org/, August 2005.

[PTOK04] Paruchuri, Praveen, Milind Tambe, Fernando Ordonez und Sarit Kraus: *Towards a Formalization of Teamwork With Resource Constraints*, 2004.

[qua06] *QuantiSpeed Architecture*. http://de.wikipedia.org/wiki/QuantiSpeed, Januar 2006.

[RLLS97] Rajkumar, R., C. Lee, J. Lehoczky und D. Siewiorek: *A Resource Allocation Model for QoS Management*. In: *IEEE Real-Time Systems Symposium*, Seiten 298–307, Dezember 1997.

[Tam97] Tambe, M.: *Towards Flexible Teamwork*. Journal of Artificial Intelligence Research, 7:83–124, 1997.

[The03] The OWL Services Coalition: *Semantic Markup for WebServices(OWL-S)*. http://www.daml.org/services/owl-s/1.0, 2003.

[TPBU05] Trumler, Wolfgang, Jan Petzold, Faruk Bagci und Theo Ungerer: *AMUN: an autonomic middleware for the Smart Doorplate Project*. Personal Ubiquitous Comput., 10(1):7–11, 2005.

[TvS03] Tanenbaum, Andrew und Marten van Steen: *Verteilte Systeme - Grundlagen und Paradigmen*. Pearson Studium, 2003.

[Web04] Web Ontology Working Group: *Web Ontology Language (OWL)*. http://www.w3.org/TR/owl-features/, February 2004.