Organic Ubiquitous Middleware

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Angewandte Informatik

der Universität Augsburg

genehmigte

Dissertation

von

Wolfgang Trumler

Erster Gutachter: Prof. Dr. rer. nat. Theo Ungerer
Zweiter Gutachter: Prof. Dr. rer. nat. Bernhard Bauer

Tag der mündlichen Prüfung: 11. Juli 2006

Zusammenfassung

Die stetig wachsende Rechenleistung und Kommunikationsbandbreite ermöglichen immer komplexere Computersysteme. Ubiquitäre Systeme, die sich durch eine Vielzahl vernetzter Computer in einer heterogenen Hard- und Softwareumgebung auszeichnen, steigern die Komplexität zusätzlich. Dies führt dazu, dass moderne Computersysteme kaum noch beherrschbar sind.

Die Initiativen des *Autonomic Computing* und des *Organic Computing* haben sich zum Ziel gesetzt, diese Komplexität beherrschbar zu machen, indem Systeme entworfen werden, die ähnliche Eigenschaften aufzeigen, wie sie auch in der Natur vorgefunden werden. Zukünftige Systeme sollen die Fähigkeit zur Selbstkonfiguration, Selbstoptimierung, Selbstheilung und zum Selbstschutz besitzen. Darüber hinaus sollen sie anpassungsfähig sein und vorausschauend handeln. Diese Attribute werden auch als Selbst-X-Eigenschaften bezeichnet.

In dieser Arbeit wird eine Middleware entworfen, mit der die Grundlage für die Umsetzung der Selbst-X-Eigenschaften für ubiquitäre Systeme geschaffen wird. Aufbauend auf der grundlegenden Architektur, die sich von herkömmlichen Middlewaresystemen in zentralen Punkten der Kommunikation und der Integration von Funktionalitäten in Form von Diensten unterscheidet, wird die Umsetzung einer Selbstkonfiguration und einer Selbstheilung erläutert und sowohl in Simulationen, wie auch in der realen Middleware evaluiert.

Die Selbstkonfiguration beruht auf dem Verhalten kooperativer sozialer Gruppen, deren Aufgabe in der Lösung eines gemeinsamen Problems liegt. Mit dem dezentralen Algorithmus werden Dienste so auf die vorhandenen Knoten eines Netzwerks verteilt, dass die Ressourcen auf den einzelnen Knoten möglichst gleichmäßig ausgelastet werden.

Die Selbstoptimierung hat das menschliche Hormonsystem zum Vorbild, das seine Informationen, die Hormone, in den Blutkreislauf ausschüttet und dadurch die Funktionen von Zellen im Körper beeinflussen kann. Die Selbstoptimierung prägt den Nachrichten der Middleware Lastinformationen des lokalen Knotens auf, die von den anderen Knoten ausgewertet werden. Anhand der Lasten anderer Knoten kann jeweils lokal entschieden werden, ob ein Dienst auf einen anderen Knoten verlegt werden soll oder nicht. Für das Verhalten des dezentralen Ansatzes ist dabei besonders das dynamische Verhalten der Diensten von Interesse, da für die Verlegung von Diensten Lastspitzen toleriert werden müssen.

Vorwort

Die vorliegende Arbeit entstand in den Jahren 2002 bis 2006 während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Informatik der Fakultät für Angewandte Informatik der Universität Augsburg. Die Middleware war ursprünglich für den Einsatz in ubiquitären Umgebungen konzipiert worden. Ab 2003 wurde die Arbeit stark durch die Gedanken des Autonomic und Organic Computing geprägt. Seit 2005 ist die Middleware Bestandteil der Forschungen im DFG-Schwerpunktprogramm 1183 „Organic Computing“ und durch die Ideen des Forschungsantrags geleitet.

Bedanken möchte ich mich bei Herrn Prof. Dr. Theo Ungerer für die vielen fruchtbaren Diskussionen und seine hervorragende Betreuung. Außerdem danke ich Herrn Prof. Dr. Bernhard Bauer für die Übernahme des Korreferats und die Hilfestellung im Bereich der Middleware- und Agentensysteme.

Mein Dank gilt weiterhin allen Mitarbeitern und studentischen Hilfskräften, die an der Umsetzung der Ideen mitgewirkt haben. Meinen Kollegen Herrn Jan Petzold und Herrn Faruk Bagci, die mit angrenzenden Themen zur Kontextvorhersage und zu mobilen Agenten die Arbeit in Bezug auf ubiquitäre Anwendungen unterstützt haben. Besonderer Dank gilt ihnen für die Unterstützung bei der Umsetzung der Ideen des Smart-Doorplate-Projekts, das teilweise als Grundlage für die Evaluationen der Middleware diente.

Darüber hinaus bedanke ich mich bei Herrn Tobias Thiemann und Herrn Robert Klaus, die mit ihren Diplomarbeiten einen Beitrag zur Umsetzung und Simulation der Selbstoptimierung und Selbstkonfiguration geleistet haben. Für die Durchsicht dieser Arbeit in Bezug auf Orthografie und Zeichensetzung danke ich Frau Erika Hambrock und meiner Frau Elke Hambrock-Trumler.

Augsburg im Juni 2006 *Wolfgang Trumler*

Inhaltsverzeichnis

1 Einleitung

Die stetig wachsende Rechenleistung moderner Computersysteme ermöglichen immer größere und komplexere Softwaresysteme. Die noch rasantere Entwicklung in der Kommunikationstechnologie erlaubt es darüber hinaus, viele Computer durch eine leistungsfähige Netzwerkverbindung miteinander zu vernetzen. Die daraus entstehenden Systeme wurden in den vergangenen Jahren so groß, dass es zum Teil mehrere Administratoren erfordert, die Systeme, die unter anderem auch unternehmenskritische Informationen zu jedem Zeitpunkt bereitstellen müssen, betriebsbereit zu halten.

Eine Entwicklung ganz anderer Art entstand Mitte der 90er Jahre, die Mark Weiser bereits 1991 vorhergesagt hatte. Die zunehmende Einbettung von Rechenleistung in alltägliche Gegenstände und deren Vernetzung untereinander ermöglicht plötzlich neue Anwendungen, die den Menschen in seinem Alltag unterstützen und von Routineaufgaben befreien können.

Beide Entwicklungen führen dazu, dass es eine wachsende Zahl an Applikationen gibt, die nicht nur auf viele Computer verteilt sind, sondern auch alltägliche Gegenstände wie Mobiltelefone und PDAs integrieren. Die Anforderungen, die an eine Middleware gestellt werden, wachsen so schnell, dass diese der technisch Entwicklung nicht mehr gerecht werden. In gleichem Maß steigt jedoch auch der Aufwand, diese technischen Systeme zu warten. Hält diese Entwicklung weiter an, sehen sich Betreiber großer Systeme bald vor einem unlösbaren Problem – der Administration und Verwaltung hoch komplexer Computersysteme.

Dieses Problem hat Paul Horn, Vizepräsident der Firma IBM, 2001 in einem Positionspapier adressiert und einen Wandel in der Entwicklung zukünftiger Computersysteme gefordert. Aufbauend auf dem Beispiel des vegetativen Nervensystems des Menschen hat er Eigenschaften zukünftiger Systeme formuliert, die helfen sollen, die Komplexität dieser Systeme beherrschbar zu machen. Zukünftige Systeme sollen nach seiner Einschätzung selbstkonfigurierend, selbstoptimierend, selbstheilend, selbstschützend, anpassungsfähig und vorausschauend sein, Eigenschaften, die das vegetative Nervensystem des Menschen erfüllt, der selbst ein hoch komplexes, aus vielen Teilen bestehendes System darstellt. Seine Vision nennt Paul Horn *Autonomic Computing*[1].

[1] Das vegetative Nervensystem wird im Englischen als autonomic nervous system bezeichnet, woraus der Name der Initiative abgeleitet wurde.

Im Rahmen eines Zukunftsworkshops der Technischen Informatik-Fachbereiche der beiden deutschen Fachgesellschaften ITG und GI wurde die Idee der *Organic Computing* Initiative entwickelt. Aufbauend auf den Ideen des Autonomic Computing, das so genannte *self-managing systems* propagiert, den Fokus also klar auf den Betrieb von Rechenzentren und Servern legt, werden die Ziele des Organic Computing weiter gesteckt und besonders eingebettete und ubiquitäre Systeme und Anwendungen in der Forschung betrachtet.

Die Organic Computing Initiative untersucht dabei auch Phänomene wie Emergenz oder Autopoiese. Anstelle von selbstverwaltenden Systemen stehen selbstorganisierende Systeme im Mittelpunkt. Selbstorganisation ist ein Prinzip, das nicht nur aus der Physik bekannt ist, sondern an vielen Beispielen in der Natur beobachtet werden kann. Sehr häufig treten diese Phänomene gerade bei komplexen Systemen auf, die aus einer großen Anzahl einzelner Elemente bestehen, wie beispielsweise Ameisenkolonien. Schwarmintelligenz und genetische Algorithmen sind weitere Themengebiete, mit denen sich Forschungsvorhaben des Organic Computing auseinander setzen. Ziel der Organic Computing Initiative ist es, diese Phänomene erklären zu können, um sie bei der Entwicklung moderner Computersysteme mit einfließen zu lassen.

Neben den Initiativen des Autonomic und Organic Computing gibt es die *Grand Challenges in Computing* [21] der British Computer Society, die zentrale Herausforderungen nennt, die in einem Zeithorizont von zehn bis fünfzehn Jahren erforscht werden sollen. Von den sieben großen Herausforderungen befassen sich zwei direkt mit den Problemen ubiquitärer Systeme. Ein weiteres Thema beschäftigt sich mit der Zuverlässigkeit zukünftiger Systeme.

Unter anderem werden für ubiquitäre Systemen Forderungen nach Selbstkonfiguration und Selbstbewusstsein genannt. Dass derartige Ziele zu den großen Herausforderungen der Wissenschaft ernannt werden, zeigt die Wichtigkeit und Bedeutung des Forschungsgebiets. Es zeigt aber auch die Dringlichkeit, mit der Lösungen für diese Probleme gefunden werden müssen.

1.1 Ziele der Arbeit

Die vorliegende Arbeit hat die Entwicklung einer Middleware für ubiquitäre Umgebungen mit dem Fokus auf die Forderungen des Organic Computing zum Ziel. Es soll eine Middleware entwickelt werden, die einerseits die Integration unterschiedlichster Geräte und Kommunikationsinfrastrukturen erlaubt, andererseits die Forderungen des Organic Computing nach Selbstkonfiguration, Selbstoptimierung, Selbstheilung und Selbstschutz unterstützt und als allgemeine Mechanismen in der Middleware verfügbar macht.

Es gibt bereits Middlewaresysteme, die für den Einsatz in ubiquitären Umgebungen entwickelt wurden. Dabei zeigt sich jedoch das Problem, dass die Verwaltung der Applikationen und die Administration der Umgebungen mit herkömmlichen Mitteln bei einer großen Zahl an Geräten nicht mehr zu bewältigen ist. Es muss ein neuer Ansatz für die Verwaltung und Administration dieser Systeme gefunden werden. Die Selbstkonfiguration und Selbstoptimierung, die im Rahmen dieser Arbeit betrachtet werden, bieten Möglichkeiten, die Verwaltung der Systeme deutlich zu vereinfachen, beziehungsweise automatisch durch das System erledigen zu lassen.

Ziel der Selbstkonfiguration ist die initiale Konfiguration des Systems anhand einer Konfigurationsspezifikation, in der die Dienste der Applikation, mit ihren Anforderungen an die vorhandenen Ressourcen, hinterlegt sind. Anhand der Konfigurationsspezifikation sollen die Dienste auf die Knoten des Netzes verteilt werden, so dass die Ressourcen der Knoten gleichmäßig ausgelastet werden. Die Selbstkonfiguration soll mit einer möglichst geringen Anzahl an Nachrichten auskommen.

Die Selbstoptimierung soll die Diensten der Applikation auf die Knoten des Netzes anhand von Lastparametern verteilen, so dass die Knoten gleichmäßig durch die Dienste belastet werden. Hierzu soll ein intensives Monitoring zur Laufzeit eingesetzt werden. Ziel der Selbstoptimierung ist, die Verteilung der Lasten mit einer minimalen Anzahl an Dienstverlegungen zu realisieren.

Herkömmliche Middlewaresysteme nutzen das Stub-/Skeleton-Verfahren für den entfernte Methodenaufrufe. Die Organic Ubiquitous Middleware verfolgt bewusst einen anderen Ansatz in der Nachrichtenvermittlung. Es soll gezeigt werden, dass eine einfache Nachrichtenvermittlung auf Basis typisierter Nachrichten flexibler ist und die Grundlagen für die Umsetzung der Selbst-X-Eigenschaften bietet.

Eine klare Trennung der Middleware von den benutzten Kommunikationsmechanismen und minimale Anforderungen an die Kommunikationsinfrastruktur soll den Einsatz der Middleware in vielen Bereichen ermöglichen.

Der Ansatz, eine Applikation in Dienste zu zerlegen ist üblich und wird insbesondere bei den WebServices verfolgt, aber die Realisierung frei verlegbarer Dienste zwischen Knoten geht über die Möglichkeiten bisheriger dienstbasierter Applikationen hinaus.

In der vorliegenden Arbeit werden die Selbstkonfiguration zur initialen Konfiguration des Systems anhand einer Konfigurationsbeschreibung und die Selbstoptimierung anhand mehrerer Optimierungskriterien mit unterschiedlichen Bewertungsfunktionen entwickelt. Weitere Selbst-X-Eigenschaften werden im Rahmen dieser Arbeit nicht untersucht, sind jedoch Gegenstand angrenzender Forschungen am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme. Unter anderem werden Ansätze zur Selbstheilung und zum Selbstschutz untersucht.

1.2 Aufbau der Arbeit

Im folgenden Kapitel werden die Forderungen der Autonomic und Organic Computing Initiativen erläutert und die Grundlagen für die Entwicklung der Organic Ubiquitous Middleware erarbeitet. Anhand verschiedener Lösungsansätze für ubiquitäre Middlewaresysteme werden die Anforderungen an die Organic Ubiquitous Middleware herausgearbeitet, die in einem Vergleich der Ansätze die Schwächen bisheriger Systeme aufzeigt.

In Kapitel 3 wird die Architektur der Organic Ubiquitous Middleware beschrieben, die als Basis für die Umsetzung der Selbst-X-Eigenschaften dient. Eine allgemeine Beschreibung der vorgeschlagenen Architektur verdeutlicht den Aufbau der Middleware, sowie die grundlegenden Konzepte beim Design. Die einzelnen Ebenen werden anschließend im Detail erklärt und die Umsetzung und Implementierung erläutert.

Kapitel 4 stellt die Selbstkonfigurationskomponente der Middleware vor. Basierend auf der Fähigkeit, Dienste auf verschiedenen Knoten auszuführen, wird ein kooperativer Lösungsansatz für eine initiale Selbstkonfiguration vorgestellt. Anhand einer vorgegebenen Konfigurationsspezifikation wird durch das kooperative soziale Verhalten der Knoten eine Anfangsbelegung erreicht, mit der die Dienste abhängig von ihren Anforderungen an die vorhandenen Ressourcen auf die Knoten verteilt werden. Dabei wird versucht, mit möglichst wenigen Nachrichten und ohne zentrale Kontrolle die Dienste auf die Knoten zu verteilen. Die Selbstkonfiguration wird anhand eines Simulators evaluiert und eine Umsetzung in der Middleware vorgestellt.

Die Selbstoptimierung, basierend auf der Idee des menschlichen Hormonsystems, wird in Kapitel 5 erklärt. Ziel der Selbstoptimierung ist die Verteilung der Dienste auf den Knoten so zu verändern, dass die betrachteten Optimierungskriterien möglichst gut erfüllt werden. Zu diesem Zweck können Dienste durch so genannte Übergabestrategien zwischen Knoten verlegt werden, so dass die betrachteten Ressourcen auf jedem Knoten ungefähr gleich stark belastet werden. Die Übergabestrategien prägen zu diesem Zweck den Nachrichten zwischen den Knoten Lastinformationen auf, ähnlich wie Hormone, die in den Blutkreislauf ausgeschüttet werden. Es soll berücksichtigt werden, dass die Selbstoptimierung mit möglichst wenig Dienstverlegungen auskommt. Die Selbstoptimierung wird sowohl in einem Simulator, als auch in der Middleware evaluiert.

In Kapitel 6 wird die vorliegende Arbeit zusammengefasst und die Ergebnisse aus den Simulationen und den Evaluationen in der Middleware werden bewertet. Das Kapitel endet mit einem Ausblick auf mögliche weitere Forschungsthemen, die aufbauend auf dieser Arbeit in die Middleware integriert werden können.

2 Grundlagen

Die Grundlagen für die Entwicklung der Organic Computing Middleware sind wegen der unterschiedlichen Forschungsrichtungen, die besonders bei der Selbstkonfiguration und der Selbstheilung einfließen, sehr weit gefächert. Die Selbstkonfiguration, basierend auf einem kooperativen sozialen Verhalten ist unter anderem auch in Agentensystemen zu finden. Die Selbstoptimierung hingegen bedient sich eines biologischen Ansatzes und hat das menschliche Hormonsystem zum Vorbild.

Es würde den Rahmen dieses Kapitels sprengen, alle Bereiche im Detail zu erläutern. Aus diesem Grund wird nachfolgend auf die Forderungen und Ansätze des Autonomic Computing und Organic Computing eingegangen und anschließend werden die wichtigsten Entwicklungen im Bereich der ubiquitären Middlewaresysteme vorgestellt. Verwandte Forschungsarbeiten im Bereich der Selbstkonfiguration und der Selbstoptimierung werden in den jeweiligen Kapiteln erläutert.

2.1 Autonomic Computing

Gordon Moore stellte 1965 die These auf, dass sich die Rechenleistung und die Speicherkapazität alle 18 Monate verdoppeln wird. Die als Moore'sches Gesetz bekannte Aussage hat bis heute Gültigkeit und wird voraussichtlich noch mindestens eine Dekade mit den herkömmlichen Fertigungsverfahren möglich sein. Die Kommunikationsbandbreite wächst sogar dreimal so schnell, was einer Verdoppelung innerhalb von sechs Monaten entspricht.

Mit der gewonnenen Rechenleistung, den verfügbaren Speicherkapazitäten und den hohen Kommunikationsbandbreiten wurden immer komplexere und leistungsfähigere Applikationen möglich. Moderne Softwaresysteme, wie beispielsweise das ERP-System der Firma SAP, besitzen eine so hohe Komplexität, dass zu deren Anpassung und Einführung viele Entwickler benötigt werden, die in ihrem speziellen Anwendungsbereich die notwendigen Einstellungen und Anpassungen vornehmen.

Vor dem Hintergrund der stetig wachsenden Komplexität und der prognostizierten Möglichkeiten hat Paul Horn im Jahre 2001, als Vizepräsident der Firma IBM, in einem Positionspapier zum Autonomic Computing [24] einen Wandel in der Entwicklung von Applikationen gefordert. Seine Vision zukünftiger Computersysteme basiert auf der Idee des vegetativen Nervensystems des Menschen.

Den Menschen als Vorbild eines komplexen Systems hat er verschiedene Forderungen aufgestellt, die ein Computersystem erfüllen muss, um in Zukunft beherrschbar zu sein. Die Systeme sollten selbstkonfigurierend, selbstoptimierend, selbstheilend, selbstschützend, umgebungsbewusst und vorausschauend sein. All diese Forderungen erfüllt das vegetative Nervensystem des Menschen ohne bewusste Kontrolle.

Wenn eine Person beispielsweise zu rennen beginnt, werden der Herzschlag und der Blutdruck erhöht und die Atmung beschleunigt. Aktionen, die nicht der bewussten Steuerung des Menschen bedürfen.

Die genannten Selbst-X-Eigenschaften sind somit Attribute eines selbstverwaltenden Systems, das sich auf verändernde Umgebungen einstellen kann. Die Steuerung durch den Administrator erfolgt nicht auf der untersten Ebene in Form von konkreten Befehlen, sondern auf einer abstrakten Ebene in Form von Zielvorgaben. Im Beispiel des Menschen war das Ziel zu rennen.

Basierend auf den Ideen von Paul Horn wurden die „Autonomic Computing Concepts“ [25] verfasst. Zwei Jahre später veröffentlichte Jeffrey Kephart in dem Artikel „The Vision of Autonomic Computing“ [30] einen Vorschlag für eine Architektur eines Autonomic Computing Systems. Darin wird von einem hierarchischen Ansatz einzelner Elemente ausgegangen, von denen jedes als Autonomic Element einen Autonomic Manager besitzt, der die Ressourcen des Elements verwaltet und anhand äußerer Vorgaben anpasst und Statusinformationen nach außen gibt.

Der Autonomic Manager stellte in jedem dieser Elemente eine übergeordnete Instanz dar, die durch Beobachtung (Monitoring) und Analyse (Analyze) der verwalteten Ressource einen Plan (Plan) erstellen und ausführen (Execute) kann. Dieser MAPE-Zyklus (Monitor-Analyze-Execute-Plan) findet sich in allen Elementen wieder und stellt die Grundlage der Architektur dar. Aus diesen wieder verwendbaren Elementen können Hierarchien entstehen, die dann als Ganzes eine Autonomic Computing Applikation darstellen.

Bereits Paul Horn hatte in seinem Positionspapier angemerkt, dass es paradox sei, ein System zunächst noch komplexer zu machen, um es dadurch zu beherrschen. Sollte jedoch die Komplexität der Systeme im selben Maß weiter anwachsen wie bisher würde dies unweigerlich zu einem Kollaps der Systeme führen, wenn diese nicht in der Lage sind sich selbst zu verwalten, da es dem Menschen nicht mehr möglich ist, die Komplexität zu beherrschen.

Allerdings ist es ein weiter Weg von heutigen Anwendungen zu Autonomic Computing Systemen. IBM schlägt in [17] eine Vorgehensweise in fünf Phasen vor. Beginnend bei den heutigen Systemen (Basic Level) werden zunächst überwachte Systeme entwickelt, die selbstständig Statusinformation bereitstellen (Managed Level), die zur manuellen Wartung und Optimierung des Systems herangezogen werden kann. In der nächsten Phase (Predictive Level) werden die gewonnenen Informationen vom System selbst ausgewertet und Vorschläge für die Vorgehens-

weise angeboten. In der vierten Phase (Adaptive Level) ist das System in der Lage, seine Ressourcen weitestgehend selbstständig zu verwalten und anhand von Service-Level-Agreements die Leistungsfähigkeit des Systems zu optimieren. In der fünften und letzten Phase (Autonomic Level) stehen selbstverwaltende Systeme, die sich anhand von abstrakten Vorgaben selbstständig verwalten und organisieren können.

Die Visionen des Autonomic Computing sind sehr stark durch die Aktivitäten und die Ausrichtung der Firma IBM geprägt. Anders als bei rein wissenschaftlich fundierten Ansätzen zielen die Forderungen und Ansätze des Autonomic Computing auf die Lösung der Probleme im Betrieb großer IT-Systeme, wie sie häufig in Firmen angetroffen werden. Die Vorgabe einer konkreten Architektur zeigt, wie eng die Umsetzung der Ziele mit den Vorstellungen der Firma IBM verbunden sind.

2.2 Organic Computing

Basierend auf den Forderungen des Autonomic Computing entstand 2004, im Rahmen eines Zukunftsworkshop der Technischen Informatik-Fachbereiche der beiden deutschen Fachgesellschaften ITG und GI, die Idee des Organic Computing, deren Ziele und Visionen zukünftiger Computersysteme in einem Positionspapier [79] dargelegt sind.

Neben den Forderungen des Autonomic Computing nach Selbstkonfiguration, Selbstoptimierung, Selbstheilung, Selbstschutz, Umgebungsbewusstsein und dem vorausschauenden Handeln, geht es beim Organic Computing darum, die zugrunde liegenden Mechanismen zu erforschen und Phänomene wie beispielsweise Emergenz [22] und Autopoiese [20] zu verstehen und zu nutzen.

Ein wesentlicher Unterschied besteht in den Zielen der beiden Initiativen. Während beim Autonomic Computing der Fokus auf „Self-managing Systems“ liegt, wird beim Organic Computing die Selbstorganisation [20] untersucht. Die Selbstorganisation ist ein natürliches Phänomen, das zunächst in der Physik durch die Gesetze der Thermodynamik beschrieben wurde, später aber auch in anderen Bereichen als elementares Prinzip erkannt wurde. Bei Insektenschwärmen und bei sozialen Gruppen zeigen sich Formen von Selbstorganisation.

Die Selbstorganisation entsteht dadurch, dass ein System verschiedene Zustände einnehmen kann, von denen manche als Attraktoren bezeichnet werden, in die das System bevorzugt übergeht. Attraktoren sind meistens Systemzustände, die einen stabilen Zustand repräsentieren. Die Selbstorganisation stellt keinen eigenen Mechanismus wie die anderen Selbst-X-Eigenschaften dar, sondern beschreibt das beobachtbare Verhalten, das durch Realisierung der Selbst-X-Eigenschaften entsteht.

In manchen Fällen entsteht aus der Selbstorganisation auch Emergenz, die dadurch charakterisiert ist, dass ein globales Verhalten beobachtet werden kann, das aus den Verhaltensmustern der Einzelteile nicht ableitbar ist. Ameisen finden beispielsweise stets den kürzesten Weg zu einer Futterquelle, indem sie auf ihrem Weg Duftstoffe absondern, dem andere Ameisen folgen.

Die Organic Computing Initiative, die auf einer wissenschaftlichen Basis aufbaut, fasst ihre Ziele bewusst deutlich weiter, da die Anwendung der Selbst-X-Eigenschaften in keinem Fall auf den reinen Serverbetrieb beschränkt sein soll. Entwicklungen des Ubiquitous Computing und der eingebetteten Systeme, sowie bereits vorliegende Forschungsergebnisse aus Bereichen der genetischen Algorithmen und Schwarmintelligenz sollen mit einbezogen werden.

Besonders Systeme aus dem Bereich des Ubiquitous Computing, bei denen sehr viele unterschiedliche Geräte in einer Anwendung zusammenwirken, müssen Mechanismen gefunden werden, mit denen die Verwaltung der Geräte vor dem Anwender verborgen werden kann und automatisiert abläuft. Wünschenswert wären Systeme, die selbstorganisierend die Aufgaben und Anliegen der Anwender unterstützen und durch pro-aktives Handeln Routineaufgaben ohne weiteres Zutun des Benutzers erledigen können.

2.3 Ubiquitäre Middlewaresysteme

Mit zunehmender Verfügbarkeit von Rechenleistung ist die Forderung nach einfacher und allgemeiner Nutzbarkeit dieser Ressource entstanden. Aus ehemals monolithischen Systemen und Anwendungen gingen verteilte, auf Komponenten basierte Applikationen hervor. Der Entwurf verteilter Anwendungen stellt die Entwickler vor neue Anforderungen, besonders in einer heterogenen Umgebung, in der unterschiedlichste Hardware und Softwareplattformen angetroffen werden.

Um die Komplexität der verteilten Programmierung vor den Entwicklern zu verbergen und die Implementierung verteilter Applikationen zu erleichtern, wurden Middlewaresysteme entwickelt, die drei wichtige Anforderungen erfüllen:

- *Abstraktion von der Netzwerkprogrammierung*
  Für den Entwickler einer Anwendung soll der Zugriff auf Komponenten und Dienste vollkommen transparent sein. Dazu ist es notwendig, eine Programmierschnittstelle anzubieten, die von den tatsächlichen Gegebenheiten der Netzwerkschicht abstrahiert.

- *Interoperabilität*
  Die Nutzung der Ressourcen, besonders in heterogenen Umgebungen, erfordert eine Middleware, die auf unterschiedlichen Plattformen und Betriebssystemen verfügbar ist. Damit ist es möglich, Dienste auf unterschiedlichen

Systemen miteinander zu verbinden. Die systemübergreifende Kommunikation und Nutzbarkeit von Diensten und Anwendungen wird als Interoperabilität bezeichnet.

- *Bereitstellung von Basisdiensten*
  Bei der Entwicklung verteilter Applikationen werden bestimmte Funktionen, wie beispielsweise das Registrieren und das Finden von Diensten, immer wieder benötigt. Zu diesem Zweck werden von allen Middlewaresystemen entsprechende Basisdienste bereitgestellt. Teilweise sind diese Dienste so stark in die Middleware integriert, dass sie bei der Entwicklung der verteilten Applikation kaum noch zu Tage treten.

Damit eine Middleware diese Anforderungen erfüllen kann, muss sie zwischen dem Betriebssystem, das die grundlegenden Funktionen zur Kommunikation mit anderen Systemen bereitstellt, und der Applikation angesiedelt sein.

Es gibt eine Reihe unterschiedlicher Middlewaresysteme die in verschiedenen Gebieten zum Einsatz kommen. Darunter sind bekannte Vertreter wie DCOM [16], CORBA [33], Java RMI [65] oder Jini [85]. Diese Systeme verfügen alle über die geforderten Eigenschaften, unterscheiden sich jedoch sehr stark in der Umsetzung.

Mark Weiser proklamierte bereits 1991 in seinem Artikel „The Computer for the Twenty-First Century“ [86] einen Wandel in der Struktur zukünftiger Computersysteme und in der Art und Weise, wie diese Systeme vom Menschen bedient werden. Mark Weiser spricht in seinem Artikel vom „Ubiquitous Computing“. Der Begriff bedeutet, dass es in der Umgebung des Menschen eine beträchtliche Kapazität an Rechenleistung in unterschiedlichster Form gibt und dass diese Rechenleistung genutzt wird, um den Menschen im Alltag zu unterstützen.

Daraus folgt eine Veränderung des Benutzerverhaltens gegenüber Computern, die nicht mehr wie bisher über einen definierten Punkt, meist Tastatur oder Maus, zu bedienen sind, sondern in einer für den Menschen natürlichen und selbstverständlichen Art und Weise. Durch die Einbettung von Rechenleistung in alltägliche Gegenstände verändert sich auch der Funktionsumfang der Gegenstände und es entstehen teilweise vollständig neue Interaktionsmuster. Beispiele ubiquitärer Umgebungen sind das „Adaptive House“[41] oder das Smart-Doorplate-Projekt [69, 70].

Zur Umsetzung der Visionen von Mark Weiser werden Middlewaresysteme benötigt, die eine Vielzahl unterschiedlichster Geräte miteinander verbinden. Dazu gehören Kleinstgeräte wie Mobiltelefone oder PDAs ebenso wie Arbeitsplatzrechner und Server. Die Bandbreite der unterschiedlichen Geräte und deren verfügbare Ressourcen stellt neue Anforderungen an die Middlewaresysteme, die für die Vernetzung und den transparenten Informationsaustausch zuständig sind. Herkömmliche Middlewaresysteme wie CORBA, DCOM oder Jini benötigen zu

viele Ressourcen, womit eine einfache Anpassung an die geänderten Anforderungen nicht möglich ist.

Nicht nur die Mannigfaltigkeit der Geräte, sondern auch deren Mobilität muss bei der Entwicklung berücksichtigt werden. Eine nahtlose Integration eintretender Geräte sollte ebenso möglich sein, wie das plötzliche Verschwinden von Geräten und Ressourcen. Die Fähigkeit der Ad-hoc-Vernetzung von Geräten ist Gegenstand vieler Forschungen aus dem Bereich der Kommunikationsinfrastrukturen und Sensornetzwerken und soll hier nicht weiter behandelt werden.

Die Probleme, die sich durch ubiquitäre Umgebungen und den darauf aufbauenden Anwendungen ergeben, werden bisher auf zwei unterschiedlichen Wegen gelöst. Es gibt technologisch getriebene Middlewaresysteme, die sich primär um die nahtlose Integration der Geräte kümmert und benutzerzentrierte Middlewaresysteme, die den Anwender in den Mittelpunkt der Entwicklung stellt. Beide Lösungsansätze haben das Problem, dass sie entweder stark technologiegetriebene Lösungen produzieren und dabei den Anwendungsaspekt vernachlässigen oder durch den starken Fokus auf den Anwender, Lösungen für einzelne Anwendungsprobleme liefern und dabei die allgemeine Nutzbarkeit in den Hintergrund rückt.

Die Organic Computing Middleware wurde mit dem Fokus auf ubiquitäre Systeme entworfen und erweitert damit das Spektrum der Anforderungen. Die genannten Middlewaresysteme sind nur teilweise in der Lage diese Anforderungen zu erfüllen, beziehungsweise wurden mit einer anderen Ausrichtung für ihren Einsatz entworfen. Aus diesem Grund werden im folgenden Kapitel Middlewaresysteme betrachtet, die sich mit der Lösung der Probleme in ubiquitären Anwendungen befassen.

2.3.1 PCOM/BASE

Das Middlewaresystem PCOM/BASE besteht aus zwei Teilen, bei denen der eine Teil (BASE [2]) die grundlegende Architektur für eine dienstbasierte Middleware liefert und der andere Teil (PCOM [1]) darauf aufbauend ein Komponentensystem zur dynamischen Anpassung für ubiquitäre Anwendungen realisiert.

BASE

Die ubiquitäre Middleware BASE realisiert durch einen Micro-Broker basierten Ansatz die Integration unterschiedlichster Geräte und Kommunikationstechnologien. Kennzeichen der Middleware ist der einheitliche Zugriff auf Dienste und gerätespezifische Funktionen, die Entkopplung der Kommunikation vom zugrunde liegenden Interoperabilitätsprotokoll, sowie die dynamische Erweiterbarkeit der

unterstützten Geräte. Bei der Entwicklung von BASE standen drei Forderungen im Vordergrund.

1. *Einheitliche Programmierschnittstelle*
   Während herkömmliche Middlewaresysteme den einheitlichen Zugriff auf entfernte Dienste in den Vordergrund stellen, versucht BASE durch zusätzliche Abstraktion von den Funktionen der Geräte in einer heterogenen Umgebung zu abstrahieren, indem Proxy-Objekte den Zugriff auf unterschiedliche Gerätefunktionen vereinheitlichen.

2. *Flexible Protokollnutzung*
   Die meisten Middlewaresysteme nutzen feste Kommunikationsmechanismen für ihren Informationsaustausch. Damit sind sie auf die Möglichkeiten der unterstützten Kommunikationsmechanismen eingeschränkt und erlauben keinen Wechsel der Protokolle während einer Kommunikation. BASE unterstützt eine Vielzahl verschiedener Kommunikationstechnologien in Form eines Plug-In-Mechanismus und erlaubt die Nutzung verschiedener Kommunikationsmechanismen innerhalb eines Nachrichtenaustauschs. Eine Antwort kann also über ein anderes Kommunikations-Plug-In versendet werden, wie die Anfrage. Damit ist es möglich während einer Kommunikation den Rückkanal zu wechseln, wenn sich die Antwort über den bisherigen Kommunikationsweg nicht mehr zustellen lässt.

3. *Anpassungsfähigkeit*
   Damit die Middleware auch zukünftige Geräte unterstützt, muss sie anpassungsfähig sein und sowohl auf kleinen Geräten wie auch auf Servern eingesetzt werden können. Darüber hinaus muss sie erweiterbar sein, um die Möglichkeiten ressourcenreicher Geräte ausschöpfen zu können.

Die Realisierung der genannten Forderungen erfolgt in BASE durch einen Micro-Broker, der durch Plug-In-Module in seiner Funktionalität erweitert werden kann. Die Idee des Micro-Broker entstammt der des Micro-Kernel, der nur einen rudimentären Funktionsumfang zum Einbinden weiterer Module anbietet und damit für unterschiedliche Anwendungsfälle durch Einbinden entsprechender Module angepasst werden kann. Die einheitliche Abstraktion der Dienste und der gerätespezifischen Funktionalitäten erfolgt über Proxy-Objekte.

In BASE erfolgt die Kommunikation wie bei den meisten Middlewaresystemen durch den entfernten Aufruf von Methoden, beziehungsweise Prozeduren. Dazu wird für jeden Dienst ein Stub generiert, der als Proxy auf dem lokalen Knoten den Aufruf auf einen entfernten Dienst kapselt. Für den Dienst wird auf dem entfernten Knoten ein Skeleton benutzt, das den generierten Methodenaufruf des Stubs in einen lokalen Methodenaufruf des Dienstes umwandelt und gegebenenfalls die Antwort an den Absender zurückliefert.

Jeder Knoten in BASE besteht aus drei Elementen. Dem InvocationBroker, dem ServiceRegistry und dem DeviceRegistry. Das DeviceRegistry verwaltet alle Geräte, die über einen Kommunikations-Plug-In erreichbar sind. Das ServiceRegistry enthält alle Dienste, die über die erreichbaren Geräte des DeviceRegistry bekannt sind.

Der InvocationBroker ist die zentrale Instanz des Micro-Broker. Er ist für das Versenden und das Zustellen einer Nachricht verantwortlich. Er verfügt über einen Thread-Pool aus dem ein Thread entnommen wird, sobald eine ausgehende Nachricht verschickt werden soll oder eine eingehende Nachricht bearbeitet werden muss. Der Thread ist so lange aktiv, bis die Nachricht bearbeitet wurde, beziehungsweise eine Antwort auf eine Anfrage eingegangen ist. Anschließend wird der Thread wieder in den Thread-Pool zur weiteren Nutzung zurückgelegt. Der Vorteil dieser Methode besteht darin, dass nicht jeder Dienst als eigener Thread implementiert werden muss, sondern im Fall einer Aktivierung einen Thread vom InvocationBroker zugewiesen bekommt.

Durch die Integration verschiedener Plug-In-Module für die Kommunikation ist BASE in begrenztem Umfang in der Lage eine Selbstkonfiguration, beziehungsweise eine Selbstoptimierung durchzuführen.

PCOM

PCOM ist eine Komponentenarchitektur basierend auf BASE. Mit PCOM ist es möglich adaptive Applikationen zu entwickeln, die aus einzelnen Komponenten aufgebaut sind. Die Interaktion zwischen Komponenten wird in Verträgen festgeschrieben, in denen auch Alternativen zur Nutzung der Adaptivität der Middleware definiert werden können. In den Verträgen können auch nichtfunktionale Parameter definiert werden, die zur Optimierung einer Applikation herangezogen werden können. Aus den Verträgen kann die Abhängigkeitsstruktur der Applikation als Baustruktur ermittelt werden.

Die Komponenten einer PCOM Applikation werden in einem PCOM-Container ausgeführt, der als BASE-Dienst implementiert ist. Eine Komponente besitzt zwei Zustände, STARTED und STOPPED. Wenn eine Komponente durch einen Container erzeugt wurde, befindet sie sich zunächst im angehaltenen Zustand. Bevor die Komponente in den aktiven Zustand überführt werden kann, müssen alle Voraussetzungen aus der Abhängigkeitsstruktur erfüllt werden. Wenn eine Komponente wieder in den Zustand STOPPED überführt wird, werden alle belegten Ressourcen freigegeben und die Komponente aus dem Container entfernt.

Es wird davon ausgegangen, dass sich das Umfeld von PCOM/BASE ständig ändert und dass die Änderungen erkannt werden können. Hat eine Komponente eine Veränderung erkannt, die dazu führt, dass die Komponente nicht mehr ausgeführt werden kann, signalisiert sie diesen Zustand an ihre übergeordnete

Komponente in der Abhängigkeitsstruktur. Diese Komponente versucht anhand der Definitionen in ihren Verträgen eine Alternative für den Dienst zu finden. Gelingt dies nicht, signalisiert dies die Komponente wiederum eine Ebene höher, bis entweder eine Adaption möglich ist und das Problem durch eine andere Komponente gelöst werden kann oder aber die Applikation nicht mehr funktionsfähig ist, wenn die Wurzelkomponente ebenfalls scheitert.

Mit dieser Vorgehensweise implementiert PCOM eine Form der Selbstheilung beziehungsweise Selbstkonfiguration. Allerdings ist dieser Mechanismus aufgrund der vorausgehenden Definition der Alternativen in den Verträgen sehr unflexibel. Das Adaptionsverhalten muss also a priori definiert werden und kann nur bedingt die tatsächlichen Gegebenheiten in die Problemlösung einbeziehen.

2.3.2 GaiaOS

Ein weiterer Vertreter der technologisch getriebenen Entwicklungen ist das Middlewaresystem GaiaOS [53, 52]. Durch Anreicherung der physikalischen Welt mit Rechenleistung und deren Vernetzung entstehen so genannte Active Spaces. Die Virtualisierung der physikalischen Objekte und deren Ressourcen ermöglicht dem Benutzer die Interaktion mit dem Active Space durch GaiaOS.

GaiaOS baut auf der Scriptsprache LuaOrb [8] auf, mit der die Kommunikation zwischen verschiedenen Technologien wie CORBA, COM und Java realisiert wird. Durch das Versenden von LuaOrb-Scripten ist GaiaOS angeblich in der Lage Rekonfigurationen durchzuführen.

Die Hauptaufgabe von GaiaOS ist die Verwaltung und Koordination der Ressourcen der physikalischen Welt, wozu eine generische Computerumgebung definiert wird. Die Architektur basiert auf dem Model-View-Controller-Konzept, die durch zwei zusätzliche Elemente, dem Coordinator und dem Adapter, erweitert wird. Der Adapter ist für die Konversion und Bereitstellung der Daten für unterschiedliche Ausgabegeräte zuständig, während der Coordinator die Aufgabe einer Metainstanz übernimmt, um sicherzustellen, dass Richtlinien und Vorgaben der Benutzer eingehalten werden.

Die Architektur sieht ein Modell mit vier Schichten vor. Die Kommunikation wird über den Unified Object Bus (UOB) realisiert. Der UOB ist für die Verwaltung der Komponenten zuständig und kann zwischen verschiedenen Komponententechnologien, wie beispielsweise CORBA oder Java Beans vermitteln und unterstützt zusätzlich die Scriptsprache LuaOrb. Für die Verwaltung der Komponenten werden vier Abstraktionen definiert: Unified Component, UOBHost, Component Container und Component Manager.

Auf dem UOB baut der Gaia Kernel mit verschiedenen Basisdiensten auf. Der Naming Service speichert für jedes Element eines Active Space, mehrere Tupel mit

jeweils einem Namen und einem Wert, der dem Namensraum des Active Space zugeordnet wird. Der Event Manager bietet eine weitere Abstraktion oberhalb der Kommunikation. Für die Verteilung von Nachrichten werden Kanäle benutzt, bei denen sich die Komponenten registrieren können. Der Discovery Service verteilt Informationen über neu entdeckte Geräte und Dienste über den Discovery Channel, wobei die Verfügbarkeit von Diensten über den Presence Channel mit einem Lease-Mechanismus gesichert wird. Der Security Service sichert den Zugriff auf Daten über verschiedene Wege. Der letzte Basisdienst von GaiaOS ist der Data Object Service, der ein typisiertes Dateisystem mit adaptivem Inhalt ortssensitiv bereitstellt.

Oberhalb des Kernels befinden sich die Gaia Services, die durch zusätzliche Funktionalität den Kernel erweitern. Das Gaia Application Model vereinheitlicht durch das erweiterte Model-View-Controller-Konzept die Entwicklung von Applikationen, die wiederum in einem Active Space Execution Environment ausgeführt werden. Das Active Space Execution Environment stellt die oberste Schicht der GaiaOS Architektur da.

GaiaOS besitzt angeblich die Fähigkeit, durch den Versand von LuaOrb-Scripten, zur Rekonfiguration. Darüber hinaus ist die Middleware jedoch nicht in der Lage eine Selbstkonfiguration oder Selbstoptimierung durchzuführen.

2.3.3 Aura/AIPIS

Ein Ansatz, bei dem der Benutzer in den Fokus der Entwicklung gestellt wird, ist das Middlewaresystem Aura [19] und die darauf aufbauende Applikation AIPIS [92]. Beide Systeme helfen dem Benutzer, indem sie seine aktuelle Tätigkeit (Task) durch die Nutzung der gerade verfügbaren Geräte möglichst gut unterstützen.

Aura

Der Name Aura ist von der Idee abgeleitet, dass nicht der Benutzer selbst, sondern eine digitale Repräsentation seiner Aufgaben und Bedürfnisse (die Aura des Benutzers) mit der vorhandenen Infrastruktur interagiert und der Benutzer nur in Ausnahmefällen durch die Middleware um Hilfe gebeten wird. Im besten Fall kann das System die Aufgabe des Benutzers, der sich in der Umgebung frei bewegen kann, ohne weitere Eingriffe des Benutzers unterstützen.

Ein Beispiel wird genannt, bei dem ein Mitarbeiter seinen Computer zuhause verlässt und in sein Büro fährt. Aura erkennt die Absicht des Benutzers und sichert die Dokumente des Benutzers, um sie in gleicher Weise auf dem Arbeitsplatzrechner zu öffnen, damit er dort an der selben Stelle weiterarbeiten kann.

Die Aufgabe eines Benutzers wird in Aura als eine Ansammlung von Dokumenten verstanden, die durch das System ortssensitiv aufbereitet werden. Abhängig von den vorhandenen Geräten wird ein Dokument in der passenden Form dargestellt.

Aura benutzt dazu einen Task Manager, der das digitale Abbild des Benutzers repräsentiert. Der Environment Manager bereitet die Umgebung, in der sich ein Benutzer befindet, entsprechend den Anforderungen des Task Managers vor und berücksichtigt die Sicherheitsanforderungen. Der Context Observer reagiert auf Veränderungen in der Umgebung und leitet seine Informationen an den Task Manager und den Environment Manager weiter, damit sich diese auf die veränderte Umgebungssituation anpassen können.

Der Task Manager ist darauf angewiesen, dass er vom Environment Manager aktuelle Informationen über den Aufenthaltsort des Benutzers und die vorhandenen Geräte in der Umgebung erhält, damit er anhand der Präferenzen des Benutzers die aktuelle Aufgabe möglichst gut verwalten und planen kann.

Der Environment Manager vergleicht ständig die Anforderungen des Task Managers mit den erreichbaren Geräten und Ressourcen der Umgebung, um die Geräte dem Benutzer zur Verfügung zu stellen, die seine Tätigkeit am besten unterstützen. Verlässt der Anwender seine Umgebung übernimmt der Environment Manager das Sichern der Dokumente, um sie in der nächsten Umgebung möglichst identisch anzuzeigen. Hierzu überwacht er die Umgebung des Anwenders, um bei einer Veränderung der vorhandenen Geräte Alternativen in Abstimmung mit dem Task Manager anbieten zu können.

Die Beschreibung von Aufgaben (Tasks) ist in Aura von besonderer Bedeutung, da alle Aktionen auf die Handlungen des Benutzers ausgerichtet sind. Herkömmliche Beschreibungssprachen für Arbeitsabläufe sind hierzu nicht ausreichend, da diese meist sehr genau auf den jeweiligen Anwendungsfall ausgelegt sind. Aus diesem Grund werden vier Forderungen an die Beschreibung von Aufgaben gestellt.

1. Aufgaben sollen in unterschiedlichen Umgebungen durchgeführt werden können.
2. Der Eingriff des Benutzers soll so gering wie möglich sein.
3. Die Handlungen des Benutzers sollen pro-aktiv unterstützt werden.
4. Die Infrastruktur soll den Anwender unterstützen und nicht einschränken.

Eine genaue Erklärung der verwendeten Beschreibungssprache wird leider nicht gegeben, es wird aber darauf eingegangen, dass besonders für die Unterstützung der pro-aktiven Handlungen auf Entscheidungsbäume zurückgegriffen wird, bei denen einerseits Handlungsalternativen eingefügt werden können, andererseits die

Alternativen anhand ihrer Wahrscheinlichkeiten bewertet und für die Ausführung ausgewählt werden.

Die Ausführungen legen den Schluss nahe, dass Aura in hohem Maße selbstorganisierend ist und sowohl Selbstkonfiguration, als auch Selbstoptimierung in Bezug auf die verwendeten Ressourcen und Geräte unterstützt. Da weder eine detaillierte Beschreibung der Mechanismen, noch eine beispielhafte Umsetzung zu finden ist, kann hierzu keine weitere Aussage getroffen werden. Beispiele in darauf aufbauenden Arbeiten (siehe AIPIS) zeigen jedoch, dass für die einzelnen Applikationen ein sehr hoher Aufwand betrieben werden muss, um die gesteckten Ziele auch nur rudimentär umsetzen zu können.

AIPIS

Das Akronym AIPIS steht für „Architecture for the Integration of Physical and Informational Spaces“. AIPIS baut auf dem Middlewaresystem Aura auf und erweitert es um vier Bestandteile, die zur Umsetzung einer Beispielanwendung benötigt werden.

Wie Aura, verfolgt auch AIPIS den Ansatz eines mobilen „Embedded Users“, dessen Aufgaben (Tasks) unterstützt werden sollen. AIPIS greift dazu auf die vorhandenen Möglichkeiten von Aura zurück und fokussiert auf die Integration der physikalischen und der digitalen Umgebungen. Eingaben unterschiedlichster Geräte werden in ein einheitliches Format überführt, so dass Applikationen die Eingaben unabhängig von der Quelle verarbeiten können.

AIPIS besteht aus den folgenden vier Komponenten:

- *Application interface service*
  Der Application interface service ermöglicht den Datenaustausch zwischen den Applikationen und deren Kontrolle untereinander. Darüber hinaus stellt er die Schnittstelle zum Benutzer dar.

- *Environment service*
  Der Environment Service bildet für Aura die Brücke zwischen dem digitalen und dem physikalischen Bereich.

- *Context-based security service*
  Der security service ist für die Auswahl beziehungsweise den Schutz von Informationen zuständig. Betritt eine nicht authentifizierte Person die Umgebung, werden sicherheitsrelevante Informationen automatisch verborgen.

- *State and versioning services*
  Diese Komponente übernimmt die Belegung und Freigabe von Ressourcen, den Transport von Informationen, sowie die Integration von Informationen anderer Applikationen in Aura.

Die Module des Application interface und Environment service ermöglichen anhand des Application input manager und des Application output manager den einheitlichen Zugriff auf Ein- und Ausgaben. Der Application input manager benutzt dazu beispielsweise Spracherkennung, Handschrift- und Gestenerkennung, Augenverfolgung und Mimikerkennung. Der Application output manager Verwaltet Geräte für die Ausgabe und die zugehörige Umgebung. Neben der visuellen Darstellung auf Displays ist auch Sprachausgabe möglich.

Die Context-based security sowie State and versioning services ermöglichen den kontextabhängigen Zugriff auf Informationen und die Verwaltung der Daten, beziehungsweise der Dokumente des Benutzers, die seiner aktuellen Aufgabe zugeordnet sind. Hierzu steuert der Resource monitor den Zugriff auf den physikalischen Raum und die darin befindlichen Geräte. Der Information monitor liefert den gleichen Schutzmechanismus für digitale Informationen wie der Resource monitor für den physikalischen Raum. Der Context monitor regelt die Sicherheitsstufe und damit die Art und den Inhalt der angezeigten Informationen, die aufgrund der aktuellen Situation eingestellt werden muss.

Die Implementierte Beispielanwendung stützt sich auf die Office-Produkte der Firma Microsoft und nutzt intensiv COM/DCOM zur Steuerung der Applikationen. Das zugrunde liegende Szenario beschreibt den Fall eines Mitarbeiters, der eine Präsentation mit unterschiedlichsten Mitteln und an verschiedenen Orten erstellen möchte.

AIPIS verfolgt den Gedanken der allgemeinen Nutzbarkeit von Informationen und physikalischen Gegenständen, schafft es jedoch nicht über eine einfache, nicht wieder verwertbare Beispielanwendung hinaus zu kommen. In dem Artikel werden die vorhandenen Probleme thematisiert und darauf hingewiesen, dass eine enorme Anstrengung notwendig ist, die Anwendung in einem heterogenen Umfeld einsetzbar zu machen. Unter anderem stellt die Konvertierung der Dokumente in die Formate unterschiedlicher Anwendungen, wie beispielsweise Microsoft Office und OpenOffice, bereits ein nahezu unlösbares Problem dar.

2.3.4 OXYGEN

Der Leitgedanke des Projekts OXYGEN [11] des MIT entstammt der Vorstellung, dass in Zukunft der Umgang mit Computern so selbstverständlich sein wird wie das Atmen der Luft, daher auch der Name OXYGEN. Es wird der Wandel von der computerzentrierten Interaktion gefordert, hin zu Kommunikationsformen, die für den Menschen selbstverständlich und natürlich sind. Damit dieses Ziel erreicht werden kann, müssen verschiedene technische Herausforderungen durch OXYGEN gelöst werden.

- *Allgegenwärtig*
  Die Middleware muss überall erreichbar sein und von überall auf dieselben Daten zugreifen können.

- *Eingebettet*
  Es muss eine vollständige Integration in die physikalische Umgebung stattfinden.

- *Nomadisch*
  Anwender und Gegenstände dürfen in ihrer Bewegungsfreiheit nicht eingeschränkt werden.

- *Anpassungsfähig*
  Das System muss in der Lage sein, auf sich verändernde Umgebungen und Benutzeranforderungen adäquat reagieren zu können.

- *Leistungsfähig*
  Die Middleware muss die vorhandenen Ressourcen unter Berücksichtigung der Benutzerbedürfnisse, der verfügbaren Rechenleistung und Kommunikationsbandbreite optimal ausnutzen.

- *Zweckdienlich*
  Die Absichten des Anwenders, wie beispielsweise „auf dem nächsten Drucker drucken“ müssen verstanden und sinnvoll umgesetzt werden.

- *Ewiglich*
  OXYGEN muss immer laufen. Komponenten können verändert oder aufgrund von Fehlern ausgetauscht werden, aber das System darf nie beendet oder neu gestartet werden.

Der technologische Ansatz von OXYGEN gliedert sich in fünf Bereiche, die jeweils für die Umsetzung eines bestimmten Teils einer gedachten Gesamtlösung notwendig sind. Dabei wird zunächst von einer spezifischen Applikation abstrahiert, bei den realen Umsetzungen zeigt sich jedoch, dass die entstandenen Lösungen teilweise so anwendungsspezifisch sind, dass nur wenig Möglichkeit der Wiederverwendbarkeit besteht.

- *Gerätetechnologie*
  Die verwendeten Geräte in OXYGEN werden in zwei Arten unterteilt. Einerseits gibt es ortsfeste, eingebettete Geräte (E21), beispielsweise Computer, Kameras, Mikrophone und große Displays, die alle über eine Netzwerkanbindung, hohe Rechenleistung und ausreichend Energie verfügen. Andererseits gibt es die tragbaren Geräte (H21) wie beispielsweise Mobiltelefone und PDAs, die sowohl in ihrer Rechenleistung als auch in ihren Energiereserven begrenzt sind und mit den E21 kommunizieren können.

- *Netzwerktechnologie*
  Die Netzwerktechnologie N21, ist für die nahtlose Verbindung zwischen den E21 und den H21 zuständig. Das Netz, gedacht als Overlay-Netzwerk über verschiedene reale Netze, ist angeblich selbstkonfigurierend. Es besitzt einen verteilten Mechanismus für die Namensvergabe von Geräten, das Auffinden von Geräten und den sicheren Zugriff auf Informationen.

- *Softwaretechnologie*
  Die Softwaretechnologie ist so ausgelegt, dass grundsätzlich von veränderlichen Umgebungen ausgegangen wird. Dazu wird eine Abstraktion zur Planung und Kontrolle eingeführt, die in der Lage ist, aufgrund erkannter Veränderungen neue Lösungsvorschläge zu erarbeiten.

- *Wahrnehmungstechnologie*
  Die Wahrnehmung, sprich Interaktion mit OXYGEN erfolgt nicht wie bei herkömmlichen Systemen mittels Tastatur und Maus, sondern über Sprache und visuelles Erkennen. Für die Erkennung der Benutzereingabe werden unter anderem Spracherkennung, Gestenerkennung, Lippen- und Augenbewegung benutzt.

- *Benutzertechnologien*
  Die Benutzertechnologien sollen den Anwender bei der Automatisierung von Abläufen unterstützen, bei der Zusammenarbeit mit anderen Personen und beim Zugriff auf eigene, allgemein verfügbare Informationen und auf Informationen anderer Personen.

Das Projekt OXYGEN verfolgt einen benutzerzentrierten Ansatz zur Lösung der Probleme des Ubiquitous Computing. Die aufgeführten Lösungsansätze werden aber leider nur sehr oberflächlich und allgemein erklärt, ohne auf die Details und Probleme der Realisierung einzugehen. Die Beschreibungen der einzelnen Technologien bewegen sich eher auf der Ebene einer Marketingbroschüre als auf der Ebene detaillierter Ausführungen wissenschaftlicher Arbeiten.

Beispiel dafür ist die Selbstkonfigurationsfähigkeit des Netzes, die zwar angesprochen wird, deren Funktionsweise und zugrunde liegenden Mechanismen jedoch nicht erläutert werden. Zum OXYGEN Projekt direkt scheint es auch leider keine Veröffentlichungen im wissenschaftlichen Umfeld, Konferenzen oder Workshops zu geben. Lediglich eine Reihe von Pressemitteilungen können auf der Homepage [56] des OXYGEN Projekts eingesehen werden.

Die einzelnen Technologien werden von unterschiedlichen Gruppen des MIT bearbeitet, die zu ihren Forschungsschwerpunkten Veröffentlichungen vorweisen können, die aber nur in Einzelfällen den Bezug zu OXYGEN herstellen. Das OXYGEN Projekt leidet unter derselben Problematik wie Aura. Die Erkenntnis, den Anwender in den Mittelpunkt der Entwicklungen zu stellen zielt auf

Applikationen mit einer erhöhten Akzeptanz, scheitert jedoch häufig an den Problemen, die in der realen Umsetzung entstehen und mit heutigen Methoden nur rudimentär oder teilweise gar nicht gelöst werden können.

2.3.5 Vergleich der Ansätze

Die vorgestellten Ansätze zur Lösung der Probleme in ubiquitären Umgebungen zeigen, dass es bei den technologisch ausgerichteten Ansätzen häufig an realen Applikationen fehlt, die auf den entwickelten Systemen eingesetzt werden können. Bei den benutzerzentrierten Ansätzen steht die Anwendung im Vordergrund, die Umsetzung der Lösungen fällt jedoch meist sehr spezifisch aus und kann damit nicht für andere Anwendungsfälle wieder verwendet werden.

Mit der Organic Ubiquitous Middleware wird weder der eine, noch der andere Ansatz verfolgt. Da beide nicht zu befriedigenden Ergebnissen führen, wird ein funktionaler Ansatz gewählt. In der Middleware werden zum einen die notwendigen Basisdienste implementiert, wie sie von anderen Middlewaresystemen her bekannt sind, darüber hinaus aber auch die Selbstkonfiguration und Selbstoptimierung. Mit diesen Mechanismen ist die Middleware in der Lage, die aufwendigen konfigurations- und Optimierungsaufgaben einer verteilten Applikation weitestgehend selbstständig zu erledigen. Besonders im Bereich der ubiquitären Systeme, mit einer Vielzahl unterschiedlichster Geräte, sind diese Eigenschaften von besonderer Bedeutung, da ein Administrator nicht mehr in der Lage ist die Komplexität des Gesamtsystems zu überschauen.

Die Anbindung unterschiedlicher Kommunikationsmechanismen ist durch einen einfachen Mechanismus möglich und aufgrund der minimalen Anforderungen mit nahezu allen aktuellen Technologien denkbar.

Der Ansatz, eine Applikation vollständig in Dienste zu zerlegen, die teilweise zwischen den Knoten verlegt werden können, ermöglicht die Dekomposition von Anwendungen in kleine wieder verwertbare Komponenten, die von unterschiedlichen Applikationen genutzt werden können.

Durch die typisierte Nachrichtenvermittlung mit einer nicht fest definierten Parameterzahl ist es möglich einzelne Dienste durch neuere Versionen mit erweitertem Funktionsumfang auszutauschen, ohne die bisherige Funktionalität einzuschränken. Alle anderen Middlewaresysteme erfordern in diesem Fall ein erneutes Kompilieren der geänderten Komponente und aller direkt davon abhängigen Komponenten, was einen enormen Verwaltungs- beziehungsweise Update-Aufwand nach sich ziehen kann. Unter Umständen muss sogar die Applikation angehalten werden bevor die Komponenten ausgetauscht werden können.

Ein direkter Vergleich der vorgestellten Middlewaresysteme ist aufgrund der unterschiedlichen Ausrichtungen und Lösungsansätze nur bedingt möglich. Trotzdem wird versucht in Tabelle 2.1 die Middlewaresysteme anhand von Kriterien

Tabelle 2.1: Vergleich der Middlewaresysteme

√ = gut beziehungsweise vollständig unterstützt ○ = nur in Ansätzen oder teilweise unterstützt	PCOM/BASE	GaiaOS	Aura/AIPIS	OXYGEN	OCM
Technologisch orientierter Ansatz	√	√			
Benutzerzentrierter Ansatz			√	√	
Funktionsorientierter Ansatz					√
Unterstützung mehrerer Kommunikationsmechanismen	√	○	○	○	√
Asynchrone Kommunikation	√				√
Asynchrones Laden von Klassen und Ressourcen					√
Entfernter Methodenaufruf	√	√	√	√	○
Nachrichtenbasierter Kommunikationsaustausch	○				√
Typisierte Nachrichten		○			√
Skalierbarkeit	○	○	○	○	√
Verlegbarkeit von Diensten	○				√
Nachrichtenweiterleitung ohne erneutes Discovery					√
Kontinuierliches Monitoring auf mehreren Ebenen					√
Selbstkonfiguration	○		○		√
Selbstoptimierung	○				√
Erweiterbarkeit um weitere Selbst-X Eigenschaften					√

zu beurteilen, die für eine Organic Ubiquitous Middleware von Bedeutung sind. Da einige der aufgeführten Merkmale erst in den folgenden Kapiteln ausführlich erläutert werden, soll für die Merkmale, die nicht offensichtlich sind eine kurze Erklärung gegeben werden, warum sie für eine Middleware von Bedeutung sind.

- *Asynchrone Kommunikation*
  Jede Kommunikationsinfrastruktur ermöglicht eine asynchrone Kommunikation, da es dem Versand eines Datenpakets ohne Rückantwort entspricht und somit die einfachste und schwächste Form der Kommunikation darstellt. Im ubiquitären Umfeld sollte jeweils von den geringsten Voraussetzungen ausgegangen werden. Eine synchrone Kommunikation benötigt bereits zusätzliche Mechanismen, die auf kleinen Sensoren möglicherweise nicht implementiert werden können.

- *Asynchrones Laden von Klassen und Ressourcen*
  Bei verteilten Applikationen, deren Dienste zur Laufzeit auf andere Knoten

verlegt werden können, muss ein Mechanismus vorhanden sein, fehlende Klassen und Daten auf den neuen Knoten laden zu können. Die meisten Systeme setzen voraus, dass die benötigten Daten auf dem Knoten vorhanden sind, was bedeuten würde, dass entweder der Administrator die Daten auf jeden Knoten kopieren muss oder das System gegebenenfalls alle Daten auf den Knoten repliziert, unabhängig davon, ob sie benötigt werden oder nicht. Beides führt zu einem erhöhten Aufwand.

- *Nachrichtenbasierter Kommunikationsaustausch*
  Der nachrichtenbasierte Kommunikationsaustausch ist, wie die asynchrone Kommunikation auch, die schwächste und einfachste Form der Kommunikation. Da bei der Implementierung nur minimale Anforderungen an die Knoten gestellt werden, wird von diesem Kommunikationsmechanismus ausgegangen.

- *Typisierte Nachrichten*
  Die meisten Middlewaresysteme benutzen entfernte Methoden- beziehungsweise Prozeduraufrufe. Diese Mechanismen erfordern bereits einen enormen Implementierungsaufwand in der Middleware. Mit typisierten Nachrichten, für die sich Dienste registrieren können, steht ein weitaus flexiblerer Kommunikationsmechanismus zur Verfügung, der es darüber hinaus ermöglicht Komponenten zur Laufzeit zu aktualisieren und deren Schnittstelle zu erweitern, ohne davon abhängige Klassen ebenfalls neu kompilieren zu müssen, wie es beim Stub/Skeleton-Prinzip der Fall ist.

- *Verlegbarkeit von Diensten*
  Der entfernte Methodenaufruf ist für viele Middlewaresysteme der Kommunikationsmechanismus, um eine Applikation in einem verteilten System zu realisieren. Die Dienste werden statisch auf einem Knoten gestartet und verbleiben dort bis zum Ende der Applikation. Sollen jedoch Eigenschaften wie Selbstoptimierung oder Selbstheilung realisiert werden, muss die Möglichkeit bestehen, Dienste von einem Knoten auf einen anderen verlegen zu können.

- *Kontinuierliches Monitoring auf mehreren Ebenen*
  Grundvoraussetzung für die Realisierung der Selbst-X-Eigenschaften ist das Sammeln von Informationen über das System selbst. Nur wenn der Zustand des Systems und der Applikationen bekannt ist, kann die Middleware adaptiv und selbstorganisierend auf eine Veränderung reagieren. Dazu bedarf es eines intensiven und kontinuierlichen Monitoring, um die notwendigen Informationen bereitstellen und weiterverarbeiten zu können.

3 Organic Ubiquitous Middleware

In diesem Kapitel wird die Architektur der Organic Ubiquitous Middleware beschrieben, die als Grundlage zur Realisierung der Selbst-X-Eigenschaften dient. Zunächst wird die allgemeine Architektur der Middleware auf einer abstrakten Ebene beschrieben. Anschließend wird die Implementierung und Umsetzung in der Middleware erläutert. Die Komponenten werden in aufsteigender Reihenfolge, beginnend bei der Transportschicht über die Nachrichtenvermittlung hin zur Dienstschicht erklärt. Auf die Komponenten des Organic Managers, dessen Aufgabe in der Koordination der Selbst-X-Eigenschaften besteht und auf die Implementierung des asynchronen Klassenladers wird am Ende des Kapitels eingegangen

3.1 Architektur

3.1.1 Architekturbeschreibung

Anwendungen auf Basis der Middleware bestehen aus einer Anzahl verschiedener Dienste, die auf mehreren Knoten verteilt sein können. Durch das Zusammenwirken der einzelnen Dienste stellt sich das verteilte System für den Anwender wie eine Applikation dar. Die Architektur der Middleware ist für alle Knoten identisch, wenngleich sich die zugrunde liegende Hardware unterscheiden kann. Ebenso kann die Verteilung der Dienste zwischen den einzelnen Knoten abhängig vom aktuellen Systemzustand unterschiedlich sein.

Die Architektur eines Knotens der Middleware gliedert sich in zwei Bereiche. Den Middlewarebereich, der für die Übertragung und die Vermittlung von Nachrichten sowie die Verwaltung der vorhandenen Dienste zuständig ist und den Bereich des Organic Managers, dessen Aufgabe in der Realisierung der Selbst-X-Eigenschaften besteht. Wie in Abbildung 3.1 dargestellt, gliedern sich beide Bereiche wiederum in mehrere Elemente.

Der Bereich der klassischen Middleware (linker Bereich) besitzt als zentrales Element den EventDispatcher, der den Versand ausgehender und die Verteilung eingehender Nachrichten übernimmt. Dienste können sich beim EventDispatcher

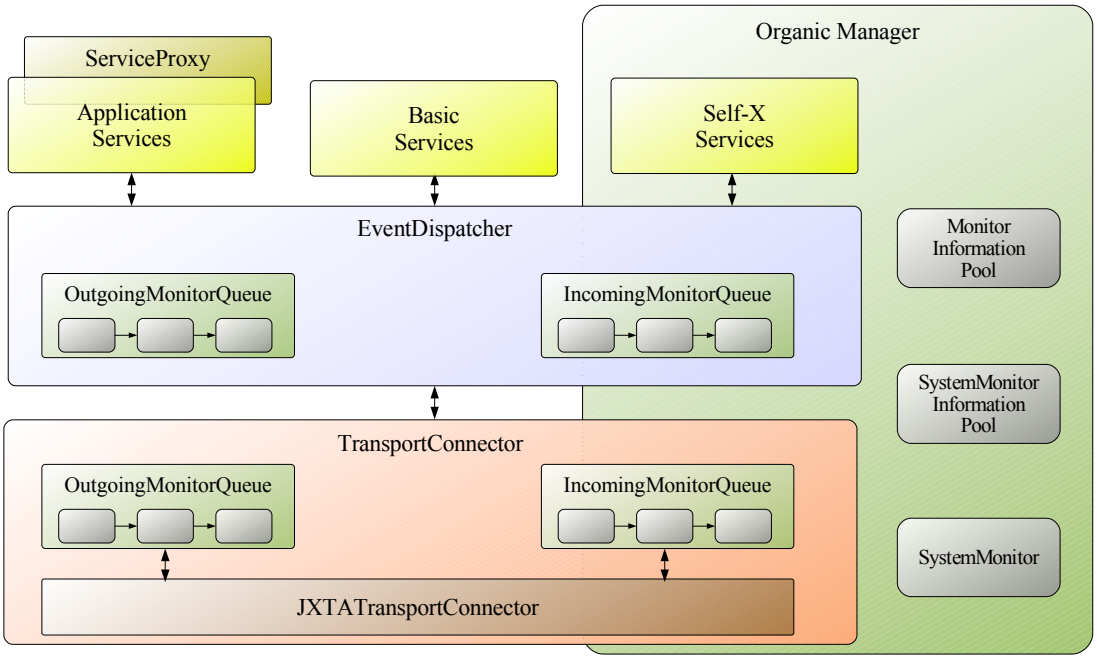

Abbildung 3.1: Organic Ubiquitous Middleware Architektur

registrieren, um sich für den Empfang von Nachrichten anzumelden. Der Nachrichtenversand erfolgt über die Transportschicht, die abhängig von der tatsächlich vorhandenen Infrastruktur durch unterschiedliche TransportConnectoren realisiert werden kann.

Der EventDispatcher und der TransportConnector besitzen jeweils eine MonitorQueue für eingehende und ausgehende Nachrichten. In einer MonitorQueue können Monitore eingebunden werden, die einerseits den Nachrichtenverkehr in der Middleware beobachten, andererseits Informationen auf die Nachrichten aufprägen, um diese an einem anderen Knoten auswerten zu können. Die Monitore bilden das Bindeglied zwischen den herkömmlichen Middlewareebenen und dem Organic Manager, indem sie Informationen im Monitor Information Pool hinterlegen können, die beispielsweise von den Selbst-X-Diensten genutzt werden können.

Der Organic Manager umfasst die Elemente zur Sammlung von Informationen aus den Monitoren, die InformationPools, den SystemMonitor, sowie die Module zur Umsetzung der Selbst-X-Eigenschaften. Die Selbst-X-Eingenschaften werden ebenfalls als Dienste in der Middleware implementiert und können in Verbindung mit entsprechenden Monitoren auf Veränderungen in der Middleware reagieren.

Die Implementierung als Dienste hat den enormen Vorteil, das die Selbst-X-Eingenschaften nicht integrale Bestandteile der Middleware sind, sondern je nach Bedarf hinzugefügt oder durch neue Versionen ersetzt werden können. Unter

Umständen könnten sogar mehrere Dienste für eine Selbst-X-Eigenschaft eingebunden werden. Entsteht dadurch eine Konkurrenzsituation zwischen den Selbst-X-Diensten, wird eine Entscheidungsinstanz benötigt, die beispielsweise durch eine Mehrheitsentscheidung oder anhand eines ausgefeilten Algorithmus die Informationen der Selbst-X-Eigenschaften weiter verwerten kann.

Nachfolgend wird zunächst der Fluss einer Nachricht durch die verschiedenen Instanzen der Middleware und die allgemeinen Aufgaben der einzelnen Schichten erläutert, bevor anschließend die Komponenten der Middleware im Detail erklärt werden.

3.1.2 Nachrichtenfluss in der Middleware

Abbildung 3.2 stellt den Fluss einer Nachricht beim Versand zwischen zwei Knoten der Middleware dar. Der Dienst auf der linken Seite verschickt eine Nachricht (`EventMessage`) und ruft dazu die Methode `sendMessage` des `EventDispatcher` auf. Der `EventDispatcher` übergibt die Nachricht intern zunächst an die `OutgoingMessageQueue`. Die `OutgoingMessageQueue` ruft auf allen eingetragenen `OutgoingMessageMonitor`-Objekten die Methode `processMessage` auf. Anschließend leitet der `EventDispatcher` die Nachricht weiter, indem er die Methode `sendMessage` beim `TransportConnector` aufruft. Der `TransportConnector` wiederum leitet die Nachricht durch die `OutgoingTransportQueue` an alle `OutgoingTransportMonitor`-Objekte und ruft danach die Methode `transportMessage` auf. Die Methode `transportMessage` muss durch einen TransportConnector, wie beispielsweise dem `JXTATransportConnector` implementiert werden, um die Nachricht auf der darunter liegenden Kommunikationsinfrastruktur verschicken zu können.

Auf der Empfängerseite kommt die Nachricht beim konkreten TransportConnector (z.B. `JXTATransportConnector`) an. Dieser ruft die Methode `processAndDispatch` auf und übergibt damit die Nachricht an seine Oberklasse, dem `TransportConnector`. Der `TransportConnector` leitet die Nachricht zunächst an die `IncomingMessageQueue` weiter, um sie durch alle Monitore zu leiten. Danach wird die Nachricht durch Aufruf der Methode `dispatchEventMessage` an den `EventDispatcher` übergeben. Bevor die Nachricht ausgeliefert werden kann, wird sie an die `IncomingMessageQueue` des `EventDispatcher` übergeben, um sie durch alle `IncomingMessageMonitor`-Objekte zu leiten. Anschließend identifiziert der `EventDispatcher` alle Dienste, die als Empfänger der Nachricht in Frage kommen und übergibt jedem eine Kopie der Nachricht.

Bei der Übergabe eines Objekts in Java wird intern lediglich ein Zeiger auf das Objekt übergeben und keine Kopie des Objekts erstellt. Damit würde jeder Dienst, an den eine Nachricht ausgeliefert wird, einen Zeiger auf das identische Nachrichtenobjekt bekommen. Würde einer der Dienste einen Wert in der Nachricht

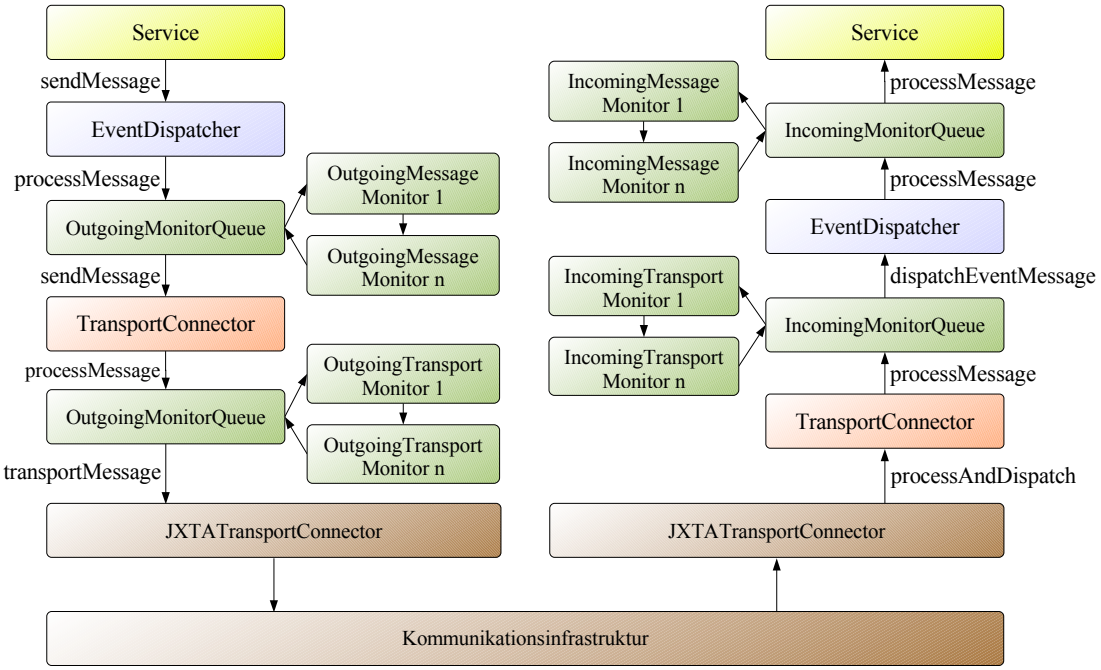

Abbildung 3.2: Nachrichtenfluss beim Versand einer Nachricht zwischen zwei Knoten

verändern, wäre diese Änderung bei allen anderen Diensten sichtbar. Um solche, meistens ungewollten Veränderungen zu vermeiden, erstellt der `EventDispatcher` für jeden Empfänger eine Kopie der Nachricht.

Werden Objekte in Nachrichten übergeben, besteht trotzdem die Gefahr von gegenseitigen Veränderungen, da Java beim Kopieren von Objekten kein Deep-Copy des Objekts erstellt. Ein Deep-Copy eines Objekts entsteht, wenn die Struktur des Objekts rekursiv kopiert wird und somit ein identischer Klon entsteht, ohne dass Teile des Objekts auf ein gemeinsames Objekt referenzieren. Die Methode `clone` in Java gewährleistet nur, dass eine Kopie von dem Objekt erstellt wird, auf dem diese Methode aufgerufen wird. Enthält das zu klonende Objekt weitere Objekte werden normalerweise nur die Referenzen auf das Unterobjekt im Klon hinterlegt. Somit ist nicht gewährleistet, dass für beliebige Objekte ein vollständiger Klon eines Objekts erstellt werden kann.

Die `clone` Methode der `EventMessage` erstellt auch Klone aller enthaltenen `MessageElement`-Objekte, womit ein vollständiger Klon einer Nachricht entsteht, sofern keine Elemente vom Typ `Object` in der Nachricht enthalten sind. Es wäre grundsätzlich möglich durch Serialisierung und Deserialisierung der Nachricht einen vollständigen Klon zu erstellen. Der damit verbundene hohe Aufwand steht jedoch in keinem Verhältnis zum möglichen Nutzen. Wird also ein Objekt in ei-

ner Nachricht verschickt, liegt es in der Verantwortung der Dienste, dieses Objekt entsprechend richtig zu benutzen und gegebenenfalls nicht zu verändern.

3.1.3 Beschreibung der Ebenen

Bevor auf die Umsetzung der einzelnen Komponenten der Middleware eingegangen wird, werden zunächst die allgemeinen Aufgaben der einzelnen Schichten erläutert, um die Zusammenhänge und Abläufe in der Middleware transparenter zu machen.

Transportschicht

Ziel der Transportschicht ist die Entkopplung der Middleware von der zugrunde liegenden Kommunikationsinfrastruktur. Darüber hinaus ist die Transportschicht für die Zustellung von Nachrichten verantwortlich. Es handelt sich immer um Nachrichten, die über eine angeschlossene Kommunikationsinfrastruktur versendet werden.

Zur Trennung der Middleware von einer spezifischen Kommunikationsinfrastruktur existiert die abstrakte Klasse `TransportConnector`. Jede konkrete Implementierung eines TransportConnectors für eine bestimmte Kommunikationsinfrastruktur erbt von der Klasse `TransportConnector`, in der die Verwaltung der Monitore enthalten ist. Die Organic Ubiquitous Middleware unterstützt das Peer-to-Peer Netzwerk JXTA und bietet dafür den `JXTATransportConnector`.

Es kann für jede beliebige andere Infrastruktur ein entsprechender TransportConnector implementiert werden, um in der Middleware zusätzliche Infrastrukturen zu unterstützen. Aufgrund der minimalen Anforderungen der Middleware ist sogar die Kommunikation über eine serielle Leitung möglich, wie sie beispielsweise beim CAN-Bus eingesetzt wird.

Nachrichtenvermittlungsschicht

Aufgabe der Nachrichtenvermittlungsschicht ist die Zustellung von Nachrichten zu den angegebenen Diensten. In der Organic Ubiquitous Middleware werden dazu typisierte Nachrichten anstelle des meist üblichen Stub-/Skeleton-Verfahrens verwendet. Dienste registrieren sich für bestimmte Typen von Nachrichten und erhalten eine Kopie einer Nachricht, wenn der Typ der eingegangenen Nachricht auf einen registrierten Typ passt.

Der `EventDispatcher` stellt die zentrale Instanz der Nachrichtenvermittlung dar und übergibt eine Nachricht an die Transportschicht, wenn die Nachricht an einen anderen Knoten gesendet werden soll, beziehungsweise übergibt die Nachricht direkt an einen Dienst, wenn ein lokaler Empfänger ermittelt werden kann.

Die Nachrichten werden bei den Diensten in einer Warteschlange hinterlegt. Mit dem Ablegen einer Nachricht, wird der Dienst über die neue Nachricht informiert und kann die eingegangene Nachricht verarbeiten.

Diensteschicht

Die Diensteschicht besteht aus drei Arten von Diensten.

- *Basisdienste*
  Zu den Basisdiensten, die direkt in der Middleware verankert sind, gehören der `ConfiguratorService`, der die Verwaltung der Dienste und Monitore übernimmt und der `DiscoveryService`, mit dem andere Knoten und Dienste auf anderen Knoten lokalisiert werden können.

- *Selbst-X-Dienste*
  Die Selbst-X-Dienste realisieren die Selbst-X-Eigenschaften der Middleware wie beispielsweise Selbstkonfiguration, Selbstoptimierung, Selbstheilung oder Selbstschutz. In der vorliegenden Arbeit wird in Kapitel 4 ein Ansatz zur Selbstkonfiguration und in Kapitel 5 zur Selbstoptimierung vorgestellt. Selbstheilung und Selbstschutz werden in dieser Arbeit nicht untersucht.

- *Applikationsdienste*
  Die Organic Ubiquitous Middleware fordert einen Wandel bei der Entwicklung von Applikationen. Weg von monolithischen Softwaresystemen hin zu Applikationen bestehend aus einer Komposition von Diensten. Die Applikationsdienste stellen die funktionalen Bestandteile der Applikationen in einzelnen Diensten bereit. Die Dienste können zur Laufzeit durch die Middleware von einem Knoten auf einen anderen verlegt werden, um die Selbstoptimierung zu realisieren. Die Rekonfiguration der Middleware bildet nicht nur die Basis für die Selbstkonfiguration, sondern auch für die Selbstoptimierung und Selbstheilung.

Organic Manager

Der Organic Manager stellt primär die Mechanismen zur Verfügung, die für das Sammeln und die eventgesteuerte Weitergabe von Informationen über den lokalen Knoten, die Dienste und die anderen Knoten benötigt werden. Der `SystemMonitor` sammelt über die lokalen Dienste Laufzeitinformationen, die für die Selbstoptimierung benötigt werden und im `SystemMointorInformationPool` hinterlegt werden. Andere Klassen können sich als Listener für verschiedene Werte anmelden. Sie werden informiert, wenn sich ein Wert ändert, für den sie sich interessieren. Der `MonitorInformationPool` stellt dieselbe Funktionalität für die Monitore des EventDispatchers und des TransportConnectors bereit.

3.2 Anwendungsbeispiel Smart-Doorplate

Um die Möglichkeiten der Organic Ubiquitous Middleware in einem realen Szenario zu demonstrieren wurde am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme das Smart-Doorplate-Projekt initiiert. Die Middleware soll im Rahmen dieses Projekts als Grundlage für verschiedene Forschungsvorhaben dienen.

3.2.1 Szenarien des Smart-Doorplates

Im Jahre 2003 entstanden die Szenarien für die Anwendung einer ubiquitären Umgebung in einem Bürogebäude. In [69] werden vier Szenarien beschrieben, die als Anwendungsbeispiele im Smart-Doorplate-Projekt umgesetzt werden sollen.

1. *Smart-Doorplate als Wegweiser*
   Eine Person betritt das Universitätsgebäude und möchte einen Mitarbeiter besuchen. Am elektronischen Empfang wählt der Besucher den Namen des Mitarbeiters aus und erhält daraufhin eine elektronische Karte anhand der er im Gebäude lokalisiert werden kann. Tritt der Besucher vor ein Türschild, erkennt die Ortsbestimmung den Besucher und veranlasst das Türschild einen Pfeil anzuzeigen in die Richtung, in der das Büro des Mitarbeiters liegt. Das Smart-Doorplate fungiert in diesem Fall als Wegweiser.

2. *Smart-Doorplate als halbtransparente Tür*
   Tritt ein Besucher vor ein Büro, kann er anhand der Informationen auf dem Türschild erkennen, ob der gesuchte Mitarbeiter anwesend ist und ob er möglicherweise gerade telefoniert. Das Türschild kann verschiedene Informationen über den Kontext des Mitarbeiters anzeigen. Anhand der verfügbaren Informationen kann der Besucher entscheiden, ob er den Mitarbeiter stören würde oder nicht. Beispielsweise könnte der Besucher so lange warten, bis der Mitarbeiter sein Telefonat beendet hat.

3. *Ein Besucher kommt an ein leeres Büro*
   Kommt ein Besucher zum Büro eines Mitarbeiters, der gerade nicht in seinem Büro ist, kann das Türschild den aktuellen Aufenthaltsort des Mitarbeiters und die Richtung dort hin anzeigen. Das Smart-Doorplate könnte auch eine Information bezüglich der Rückkehr des Mitarbeiters bekannt geben, damit der Besucher entscheiden kann, ob er warten oder eine Nachricht hinterlassen möchte.

4. *Rückkehr eines Mitarbeiters*
   Kommt ein Mitarbeiter in sein Büro zurück, kann ihn das Türschild beispielsweise darüber informieren, wie viele Besucher während seiner Abwesenheit da waren, oder welche Nachrichten für ihn hinterlassen wurden.

In den vier Szenarien finden sich drei Forschungsgebiete wieder, an denen am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme geforscht wird. Erstens die Middleware für ein ubiquitäres Umfeld, zweitens die Kontextvorhersage, die auf Basis von Bewegungsmustern einer Person den nächsten Aufenthaltsort einer Person vorhersagen kann und drittens mobile Agenten, mit deren Hilfe vertrauenswürdige Informationen im System zwischen den Knoten transportiert werden können.

3.2.2 Umsetzung der Szenarien

Die Dienste, die für das Smart-Doorplate-Projekt implementiert wurden, sind in Abbildung 3.3 an der Stelle der Anwendungsdienste abgebildet. Daneben befinden sich die Basisdienste ConfiguratorService und DiscoveryService. Da noch nicht alle Szenarien umgesetzt werden konnten, wird nachfolgend nur auf die Dienste eingegangen, die bisher in der Smart-Doorplate-Applikation implementiert wurden.

Das Smart-Doorplate-Projekt besteht im Wesentlichen aus den elektronischen Türschildern, die auf dem Flur des Lehrstuhls anstelle der normalen Türschilder installiert wurden und einem Location-Tracking-System auf Basis von Funkmodu-

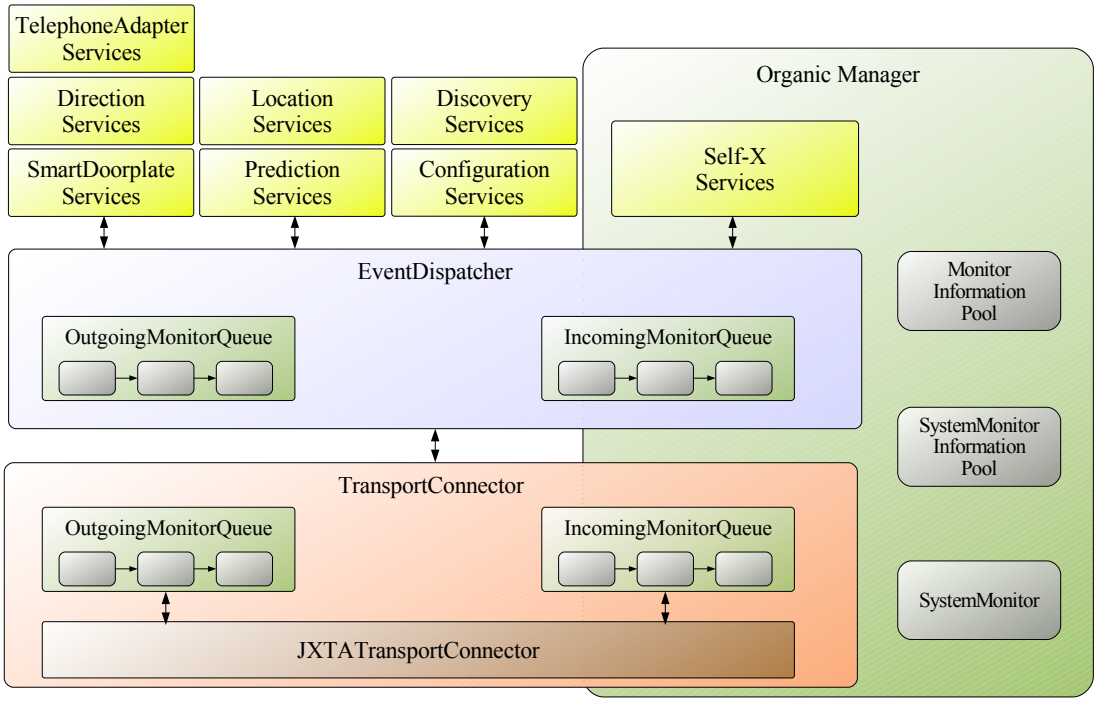

Abbildung 3.3: Dienste des Smart-Doorplate-Projekts auf einem Knoten

len. Mit den Funkmodulen ist die Ortsbestimmung der Mitarbeiter innerhalb der Räume möglich. Wird von den Modulen erkannt, dass eine Person ihre Position verändert hat, wird eine entsprechende Nachricht an die Türschilder verschickt. Der LocationService kapselt die Funktionalität der Ortsbestimmung und ist für das Versenden der Nachrichten zuständig.

Abbildung 3.4 zeigt ein Türschild bei dem beide Personen anwesend sind, erkennbar an den Symbolen vor den Namen. Fehlt das Personensymbol vor dem Namen wird damit angezeigt, dass die Person nicht anwesend ist und sich möglicherweise in einem anderen Raum befindet.

Der SmartDorplateService ist für die Verarbeitung der Positionsinformation von Personen und die Anzeige der Personensymbole zuständig. Der SmartDoorplateService erhält beim Start die Informationen über die Raumnummer und die Mitarbeiter, die in diesem Büro arbeiten. Wird eine Positionsnachricht des LocationService empfangen, entscheidet das Türschild, ob das Symbol der Person angezeigt werden soll oder nicht.

Mit dem TelephoneAdapterService werden die Telefone der Büros per TAPI an das System angeschlossen. Der Zustand des Telefonhörers wird einmal pro Sekunde ermittelt. Wird erkannt, dass ein Telefonhörer abgenommen wurde, wird eine Nachricht an das Smart-Doorplate gesendet, in der die Information über den Benutzer, der gerade telefoniert, übermittelt wird. Der SmartDoorplateService zeigt darauf hin einen Telefonhörer neben dem Symbol des Mitarbeiters an. In Abbildung 3.5 ist das entsprechende Türschild abgebildet.

Wird neben dem Namen der Person kein Symbol angezeigt, soll dies andeuten, dass die Person nicht im Raum ist. Durch Auswahl des Namens, die elektroni-

Abbildung 3.4: Das Smart-Doorplate für einen Raum mit zwei Mitarbeitern

Abbildung 3.5: Das Smart-Doorplate zeigt an, wenn ein Mitarbeiter telefoniert

schen Türschilder verfügen über einen Touch-Screen, wird angezeigt, wo sich eine Person momentan befindet, sofern die Position der Person dem LocationService bekannt ist. Abbildung 3.6 zeigt die Informationen, die das Smart-Doorplate für eine Person anzeigt.

Wenn die Position einer Person bekannt ist, wird die entsprechende Raumnummer angezeigt und die Richtung, in der dieser Raum erreicht werden kann. Die

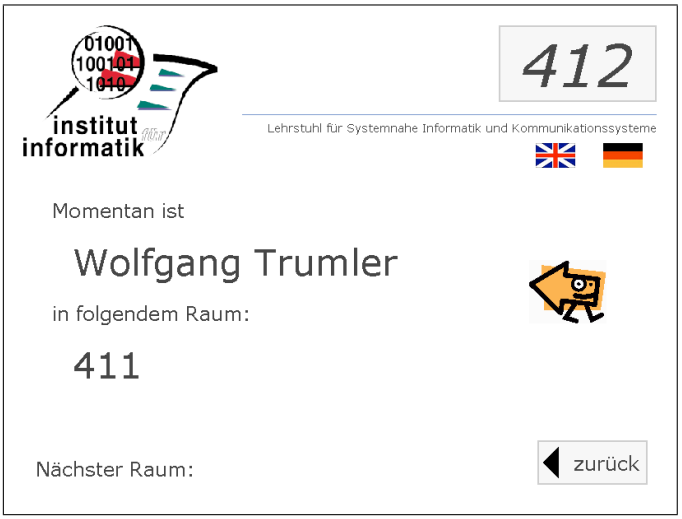

Abbildung 3.6: Das Smart-Doorplate zeigt den Aufenthaltsort einer Person und die Richtung zu dem entsprechenden Raum an

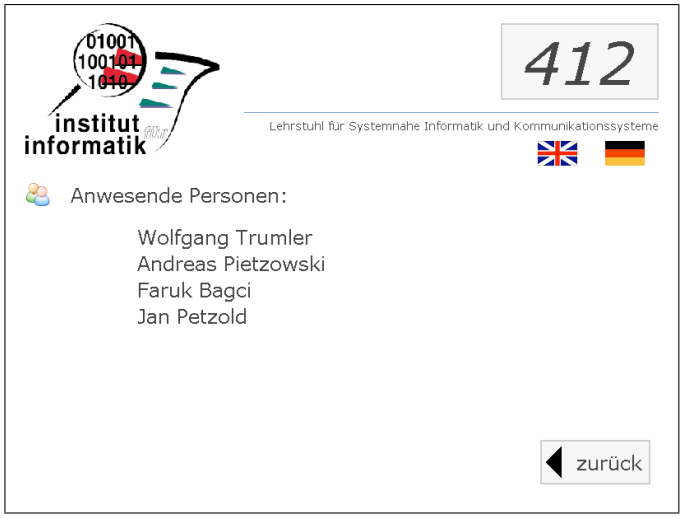

Abbildung 3.7: Das Smart-Doorplate zeigt die anwesenden Personen in einem Raum

Richtungen sind nicht in jedem Türschild fest hinterlegt, sondern werden bei Bedarf vom DirectionService erfragt.

Der DirectionService baut sich aus den Nachbarschaftsinformationen der Türschilder eine Art Landkarte auf, aus der die Richtungen zwischen den Türschildern ermittelt werden können. Jedes Türschild kennt die Raumnummern seiner angrenzenden Räume, die beim Start eines SmartDoorplateService dem DirectionService mitgeteilt werden.

Der PredictionService ergänzt die Anwendung um die Vorhersage der zukünftigen Position einer Person. Wird hinter dem Text „Nächster Raum:“ eine Raumnummer angezeigt, gibt diese die nächste vorhergesagte Position des gesuchten Mitarbeiters an. Außerdem wird ein Zeitraum angegeben, wann sich die Person dort befinden wird.

Die Informationen des PredictionService beruhen auf der Beobachtung des Benutzerverhaltens. Anhand der Bewegungsinformationen der Mitarbeiter wird versucht über geeignete Vorhersagetechniken die nächste Position zu bestimmen. Ist die Vorhersage zu unsicher, wird keine Vorhersage angeboten. Weitere Informationen zu den Forschungen bezüglich Kontextvorhersage sind in [46, 49, 47, 48] zu finden.

Eine weitere Information, die das Türschild zur Verfügung stellt, sind die Personen, die sich in einem Raum befinden. Wird die Raumnummer ausgewählt, zeigt das Smart-Doorplate die Namen der Personen, die sich gerade in diesem Raum befinden. Abbildung 3.7 zeigt dies für den Raum 412.

3.3 Implementierung der Organic Ubiquitous Middleware

Der Rest dieses Kapitels beschreibt die Umsetzung und Implementierung der Organic Ubiquitous Middleware. Es werden die einzelnen Ebenen im Detail besprochen und relevante Aspekte der Implementierung erläutert.

3.3.1 Transportschicht

Die Transportschicht bildet den Anschluss der Middleware an eine zugrunde liegende Kommunikationsebene für den Nachrichtenaustausch. Um die Middleware von den realen Gegebenheiten der Transportschicht zu entkoppeln, wird der `TransportConnector` definiert. Ein TransportConnector muss diese abstrakte Klasse implementieren, um Nachrichten zwischen den Knoten der Middleware austauschen zu können. Die Entkopplung vom eigentlichen Nachrichtenversand ermöglicht einen breiten Einsatz der Middleware bei wechselnden Kommunikationsinfrastrukturen.

Als beispielhafte Implementierung eines TransportConnectors wird der `JXTATransportConnector` bereitgestellt. Der `JXTATransportConnector` stellt eine Implementierung auf Basis des JXTA-Peer-to-Peer-Netzwerkes [50] bereit. Denkbar wären auch Implementierungen über eine serielle Schnittstelle oder einem Bussystem, wie beispielsweise dem CAN-Bus. Abbildung 3.8 zeigt die Klassen der Transportschicht.

TransportConnector

Die abstrakte Klasse `TransportConnector` definiert die notwendigen Methoden, die von einem TransportConnector bereitgestellt werden müssen. Dazu gehören die Methoden `init`, `start`, `stop` und `destroy` sowie die Methode `transportMessage`. Die Methode `init` wird direkt nach dem Erzeugen des TransportConnectors aufgerufen, um notwendige Initialisierungen durchführen zu können. Die Methode `start` wird nach der Initialisierung aufgerufen, um dem TransportConnector mitzuteilen, dass er für den Nachrichtenaustausch bereitstehen muss. Der TransportConnector kann in dieser Methode gegebenenfalls die Verbindung zur Kommunikationsschicht herstellen. Mit der Methode `stop` wird dem TransportConnector mitgeteilt, dass bis auf weiteres keine Kommunikation erfolgen soll. Der TransportConnector kann diese Methode benutzen, um vorhandene Verbindungen oder belegte Ressourcen freizugeben. Die Methode `destroy` wird aufgerufen, bevor die Instanz des TransportConnectors endgültig entfernt wird. In dieser Methode kann der TransportConnector noch belegte Ressourcen freigeben.

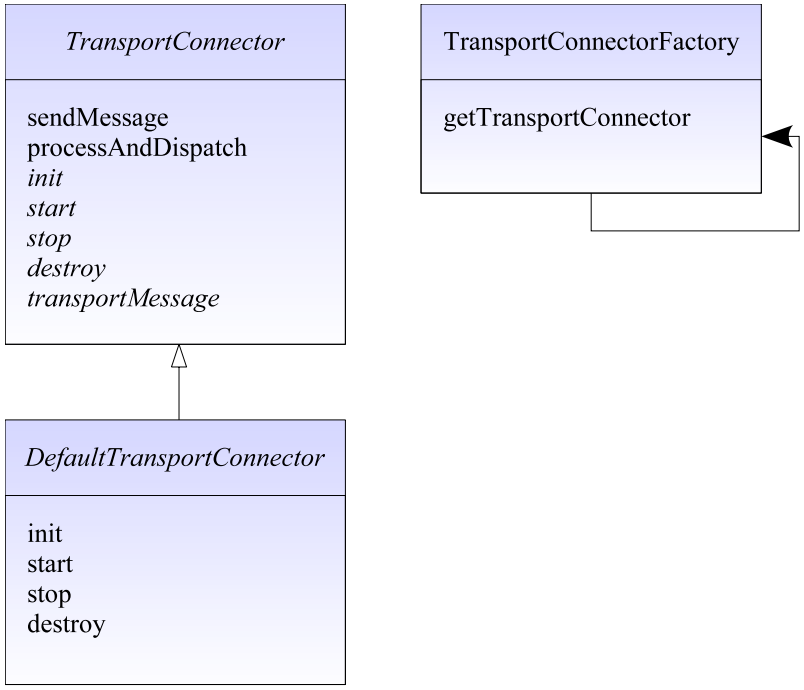

Abbildung 3.8: Klassendiagramm der Transportschicht

Die Methoden `start` und `stop` stellen komplementäre Funktionen für den Beginn beziehungsweise das Ende der Kommunikation eines TransportConnectors dar. Ebenso die Methoden `init` und `destroy`, die zum Belegen und Freigeben von Ressourcen oder Verbindungen benutzt werden können.

Wenn eine Nachricht verschickt werden soll, ruft der `EventDispatcher` die Methode `sendMessage` der abstrakten Klasse `TransportConnector` auf. Die Nachricht wird dann an die `OutgoingMessageQueue` übergeben und durch alle Monitore vom Typ `OutgoingTransportMonitor` geleitet. Anschließend wird die Nachricht an die Methode `transportMessage` des konkreten TransportConnectors (z.B. `JXTATransportConnector`) zum Versand übergeben. Anhand der Angaben im Kopf der Nachricht über den Empfänger-Knoten und Empfänger-Dienst kann der TransportConnector entscheiden, an wen eine Nachricht versendet werden muss.

DefaultTransportConnector

Der `DefaultTransportConnector` stellt eine Implementierung der abstrakten Klasse `TransportConnector` dar. Die abstrakten Methoden `init`, `start`, `stop` und `destroy` sind mit leeren Methodenrümpfen implementiert. Die Methode `transportMessage` wird nicht implementiert und muss weiterhin von einem

konkreten TransportConnector implementiert werden. Da nicht alle abstrakten Methoden der Klasse `TransportConnector` implementiert werden, ist der `DefaultTransportConnector` wiederum eine abstrakte Klasse.

TransportConnectorFactory

Um die Middleware von der Implementierung unterschiedlicher TransportConnectoren zu trennen, wird eine Fabrik (Factory) für die Instantiierung der TransportConnectoren eingesetzt. Die Klasse `TransportConnectorFactory` ist als Singleton realisiert und besitzt nur die Methode `getTransportConnector` zum Erzeugen eines TransportConnectors eines bestimmten Typs. Die Definition der TransportConnectoren erfolgt in der Datei `TransportConnectorFactory.properties`, die beim Erzeugen der `TransportConnectorFactory` gelesen wird. In dieser Datei wird jeweils einem symbolischen Namen die Klasse eines TransportConnectors zugeordnet. Der Standardeintrag für den `JXTATransportConnector` lautet:

```
JXTA = de.uau.sik.amun.transport.JXTATransportConnector
```

Die `TransportConnectorFactory` erzeugt somit eine Instanz des `JXTATransportConnector`, wenn die Methode `getTransportConnector` mit dem Parameter `JXTA` aufgerufen wird. Mit dieser Vorgehensweise ist es möglich, die Middleware um neue TransportConnectoren zu erweitern, ohne die Middleware selbst ändern zu müssen.

3.3.2 JXTATransportConnector

Der `JXTATransportConnector` wird als Standardimplementierung des `TransportConnector` mit der Middleware bereitgestellt. Basierend auf dem JXTA-Peer-to-Peer-Netzwerk wird eine Kommunikationsinfrastruktur angeboten, die sowohl Punkt-zu-Punkt als auch Broadcast-Kommunikation anbietet.

Der `JXTATransportConnector` nutzt die `Pipes` von JXTA zur Realisierung der Kommunikationswege zwischen den Knoten. Jeder Knoten erzeugt eine `BroadcastPipe` sowie mehrere `UnicastPipe`. Die `BroadcastPipe` wird benutzt, um Nachrichten per Broadcast an alle Knoten zu schicken. Die `UnicastPipe` repräsentiert eine Punkt-zu-Punkt-Verbindung zwischen zwei Knoten der Middleware. Zu jedem Paar von Knoten kann es eine `UnicastPipe` geben, über die beide Knoten direkt adressierte Nachrichten austauschen können. Die `UnicastPipes` zwischen den einzelnen Knoten werden nur auf Bedarf geöffnet. Würde von jedem Knoten zu jedem anderen Knoten eine `UnicastPipe` aufgebaut, entstünde ein vollständig vermaschtes Netz von Knoten.

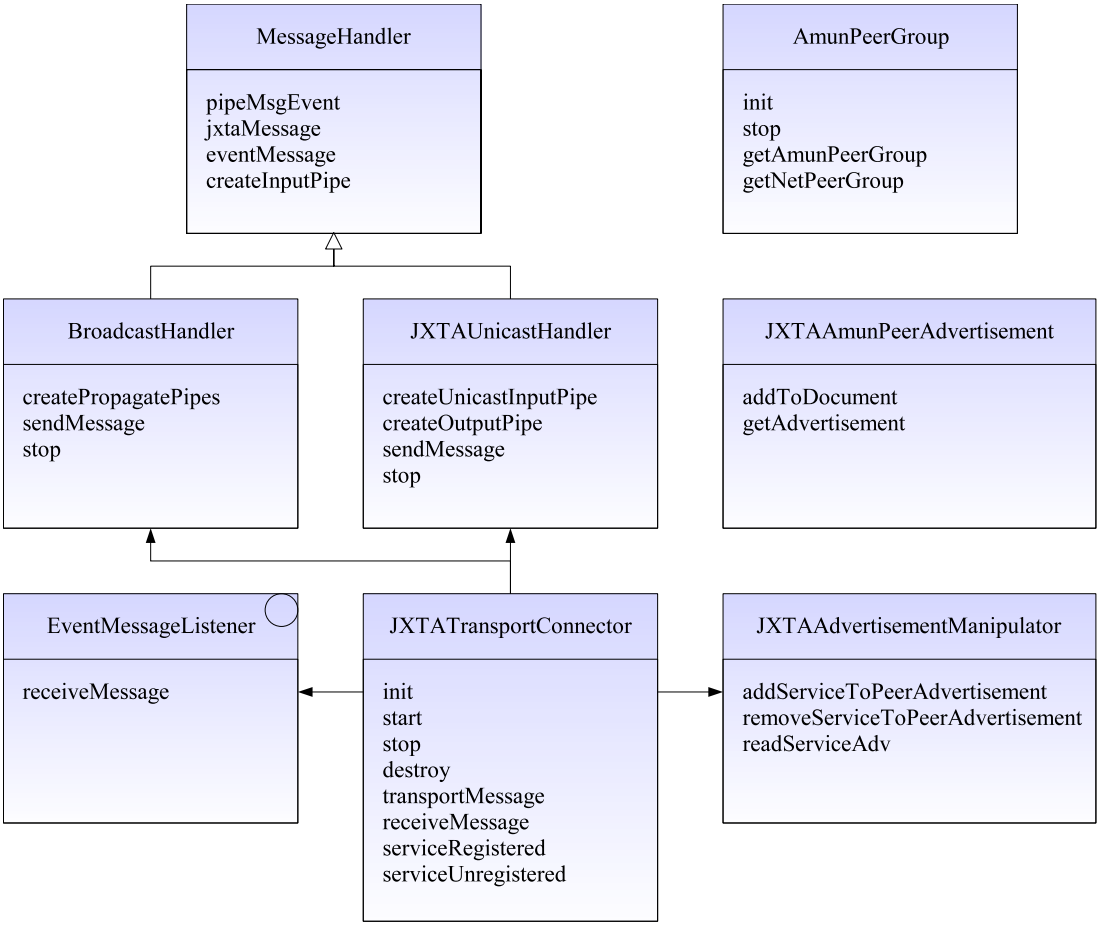

Abbildung 3.9: Klassendiagramm des JXTATransportConnectors

Abbildung 3.9 zeigt die Klassen des `JXTATransportConnector`. Die eigentliche Kommunikation mit dem JXTA-Peer-to-Peer-Netzwerk übernimmt der `MessageHandler`. Er beinhaltet die Methoden zum Erzeugen einer InputPipe, um Nachrichten empfangen und weiterleiten zu können. Mit der Methode `createInputPipe` erzeugt der `MessageHandler` eine Broadcast- oder Unicast-InputPipe, über die Nachrichten vom Netzwerk empfangen werden können. Mit der Methode `jxtaMessage` wird eine `EventMessage` für den Versand verpackt und in eine Nachricht für das JXTA-Netzwerk umgewandelt. Die Methode `eventMessage` entpackt aus einer eingegangenen JXTA-Nachricht die enthaltene `EventMessage`. Die Methode `pipeMsgEvent` wird aufgerufen, wenn eine Nachricht empfangen wurde. In dieser Methode wird aus der JXTA-Nachricht die enthaltene `EventMessage` mit der Methode `eventMessage` entpackt und an den registrierten Listener übergeben. Als Listener wird der `JXTATransportConnector` eingetragen, der dazu die Schnittstelle `EventMessageListener` implementiert. Der

`JXTATransportConnector` leitet die empfangene Nachricht an den `TransportConnector` weiter.

Der `BroadcastHandler` erweitert den `MessageHandler` und ist für die Kommunikation von Broadcast-Nachrichten über eine `BroadcastPipe` zuständig. Damit jeder Knoten im Netzwerk eine Broadcast-Nachricht empfangen kann, muss jeder Knoten über eine BroadcastInputPipe mit der gleichen Identifikation verfügen. Der `BroadcastHandler` definiert eine eindeutige Identifikation, die zum Öffnen der BroadcastPipes benutzt wird. Der `BroadcastHandler` öffnet beim Starten mit der Methode `createPropagatePipe` eine BroadcastOutputPipe und kann mit der Methode `sendMessage` über diese BroadcastPipe eine Nachricht an alle anderen Knoten im Netz schicken. Mit der Methode `stop` wird er `BroadcastHandler` veranlasst, seine Pipes zu schließen.

Nachrichten, die direkt zwischen zwei Knoten versendet werden sollen, werden über den `JXTAUnicastMessageHandler` verschickt. Der `JXTAUnicastMessageHandler` erzeugt mittels des `MessageHandler` eine `InputPipe` über die er Nachrichten empfangen kann. Der Versand einer Nachricht zu einem gegebenen Knoten wird über eine `UnicastPipe` zu diesem Knoten realisiert. Dazu besitzt der `JXTAUnicastMessageHandler` eine `HashMap`, in der für jeden Knoten, zu dem bereits eine Nachricht gesendet wurde, die zugehörige `OutputPipe` hinterlegt ist. Wird in dieser `HashMap` für einen Knoten, an den eine Nachricht gesendet werden soll, keine `OutputPipe` gefunden, erzeugt der `JXTAUnicastMessageHandler` eine neue `OutputPipe` zu diesem Knoten. Beim Versand einer Nachricht prüft der `JXTAUnicastMessageHandler` also, ob für die NodeID der Nachricht die benötigte `OutputPipe` bereits vorhanden ist und erzeugt gegebenenfalls eine neue `OutputPipe`. Wenn die Methode `stop` aufgerufen wird, schließt der `JXTAUnicastMessageHandler` alle offenen Pipes.

Die Klasse `AmunPeerGroup` kapselt die Funktionalität, mit der ein Knoten des JXTA-Netzwerks aufgebaut werden kann. Wird eine Instanz dieser Klasse erzeugt, befindet sich der Knoten zunächst in der `NetPeerGroup`. Die `NetPeerGroup` ist die oberste Gruppe JXTA-Netzwerks, in die jeder Knoten eingebunden wird, sofern er nicht explizit einer anderen Gruppe beitritt. Gruppen in JXTA stellen einen Rahmen für die Verbreitung von Nachrichten dar. Broadcast-Nachrichten werden beispielsweise nur innerhalb einer Gruppe verschickt. Um die Knoten der Middleware in eine geschlossene Gruppe aufzunehmen, erzeugt die Methode `init` die Gruppe `AmunPeerGroup` mit einer eindeutigen Gruppenidentifikation. Um der erzeugten Gruppe beitreten zu können, bedarf es einer Authentifizierung des Knotens gegenüber der Gruppe. Mit dieser Authentifizierung wird geprüft, ob der Knoten berechtigt ist der Gruppe beizutreten. Somit können nur die Knoten einer Gruppe beitreten, die auch über eine entsprechende Authentifizierung verfügen.

Damit ein Knoten innerhalb einer Gruppe (PeerGroup) gefunden werden kann, muss er ein PeerAdvertisement erstellen, in dem der Knoten Informationen über

sich selbst und über die Dienste auf diesem Knoten publiziert. Die Middleware benutzt bereits selbst ein `AmunNodeAdvertisement`, um die Informationen eines Knotens innerhalb der Middleware zu publizieren und zu verarbeiten. Das Klassendiagramm mit den verschiedenen Advertisementarten ist in Abbildung 3.13 dargestellt.

Das `JXTAAmunPeerAdvertisement` ist von der Klasse `AmunNodeAdvertisement` abgeleitet und erweitert diese um die Informationen, die für das JXTA Peer-to-Peer Netzwerk benötigt werden. Bei dem `JXTAAmunNodePeerAdvertisement` handelt es sich um ein XML Dokument, das zwischen den Knoten ausgetauscht wird. Wenn beispielsweise ein neuer Dienst dem Knoten bekannt gemacht werden soll, wird das zugehörige `ServiceAdvertisement` dem `JXTAAmunPeerAdvertisement` durch den `JXTAAdvertismentManipulator` hinzugefügt. Anschließend kann das erweiterte `JXTAAmunPeerAdvertisement` publiziert, beziehungsweise durch den Discovery Service von JXTA erfragt werden. Der `JXTAAdvertismentManipulator` kann das `ServiceAdvertisement` eines neuen Dienstes einem `JXTAAmunPeerAdvertisement` hinzufügen, ein vorhandenes entfernen und ein `ServiceAdvertisement` aus dem XML-Dokument eines `JXTAAmunPeerAdvertisement` extrahieren.

3.3.3 Nachrichtenvermittlungsschicht

Die Nachrichtenvermittlung der Middleware stellt den Austausch von Informationen zwischen den Diensten auf Basis einer asynchronen Nachrichtenvermittlung sicher. Zum Austausch von Informationen erzeugen Dienste Nachrichten-Objekte (`EventMessage`), die mit Nachrichtenelementen (`MessageElement`) gefüllt werden.

Im Vergleich zu anderen Middlewaresystemen basiert die Benachrichtigung nicht auf dem Stub/Skeleton-Prinzip [65], um direkte Methodenaufrufe auf entfernten Objekten ausführen zu können, sondern auf dem Prinzip des asynchronen Nachrichtenaustausches typisierter Nachrichten. Der asynchrone Nachrichtenaustausch hat den Vorteil, dass Dienste nicht zwingend eine vorgegebene Schnittstelle implementieren müssen, um als Dienstgeber in Frage zu kommen. Beim Stub/Skeleton-Prinzip wird dies durch die Definition der Schnittstelle in Form eines Skeletons fest vorgeschrieben. Dies bedeutet auch, dass bei einer Änderung der Schnittstelle, beispielsweise einer Erweiterung um einen optionalen Parameter, alle Dienste und Dienstnehmer neu übersetzt werden müssen, da sonst Laufzeitfehler im System auftreten.

Die Forderung nach einer asynchronen Kommunikation begründet sich in den minimalen Anforderungen, die sich daraus an eine Kommunikationsinfrastruktur ergeben und ermöglicht somit die Anbindung sehr vieler Infrastrukturen bis hin zur seriellen Kommunikation. Ein weiterer Vorteil liegt darin, dass Dienste nicht

blockiert werden, solange sie auf die Antwort einer Anfrage warten. Beim entfernten Methodenaufruf werden Objekte normalerweise blockiert bis entweder eine Antwort vom Empfänger der Nachricht gesendet wurde oder die Übertragung aufgrund eines Fehlers oder Timeouts abgebrochen wurde. Dies kann zu sehr langen Verzögerungen bei der Ausführung eines Dienstes und somit zu langen Antwortzeiten führen.

Bei der asynchronen Kommunikation können die Dienste eingehende Nachrichten verarbeiten, auch wenn noch keine Antwort auf eine versendete Anfrage eingegangen ist, da der Dienst nach dem Versand nicht blockiert wird, sondern normal weiterarbeitet. Der Nachteil der asynchronen Kommunikation liegt in der Art, wie Nachrichten verarbeitet werden müssen. Unter Umständen ist die Bearbeitung einer Nachricht mit einem Zusatzaufwand verbunden, um die Zuordnung zwischen Anfrage und Antwort herstellen zu können. Dieses Problem lässt sich jedoch beispielsweise mit Nachrichtennummern lösen.

Herkömmliche Middlewaresysteme beruhen auf der Annahme, dass eine gesicherte Kommunikation zugrunde liegt, meist TCP/IP. Dies bedeutet, dass eine Nachricht, die versendet wird entweder beim Empfänger korrekt ankommt oder eine entsprechende Fehlersignalisierung beim Sender erzeugt wird. Besonders im ubiquitären Umfeld sollte davon ausgegangen werden, dass aufgrund der sich schnell ändernden Umgebung manche Dienste nicht mehr oder aufgrund äußerer Einflüsse zeitweise nicht erreichbar sind. Aus diesem Grund fordert die Organic Ubiquitous Middleware bewusst keine gesicherte Kommunikation, um die Auswirkungen einer ungesicherten Übertragung im Zusammenspiel der Dienste, Applikationen und Selbst-X-Eigenschaften zu erforschen.

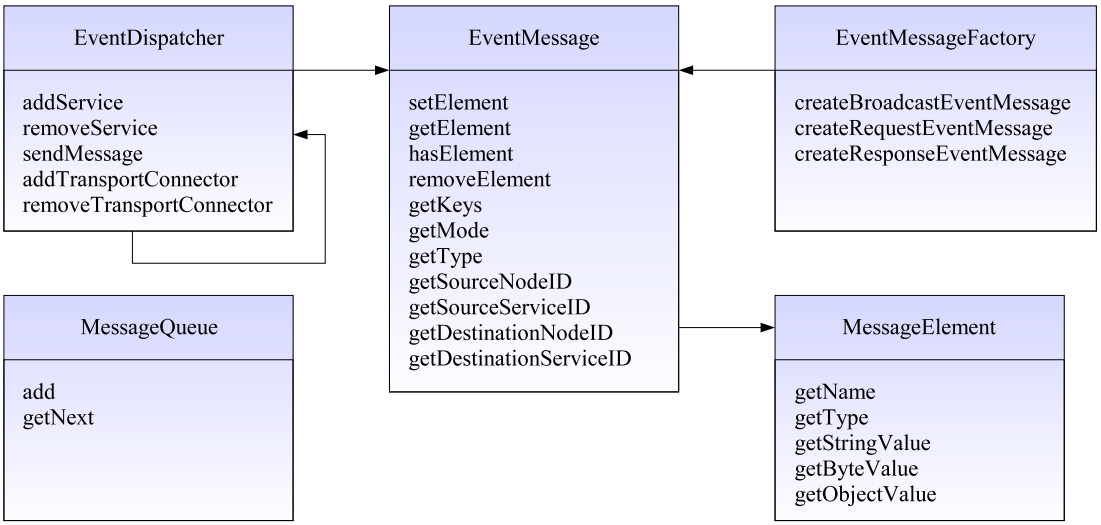

Abbildung 3.10: Klassendiagramm der Nachrichtenvermittlung

In Abbildung 3.10 ist das Klassendiagramm der Nachrichtenvermittlung abgebildet. Eine `EventMessage` kann mittels der `EventMessageFactory` erzeugt werden. Der `EventMessage` werden dann Informationen in Form von `MessageElement`-Objekten hinzugefügt. Die Nachricht wird durch den `EventDispatcher` verschickt und bei den entsprechenden Diensten in ihre `MessageQueue` gelegt.

MessageElement

Die kleinste Einheit einer Nachricht ist ein `MessageElement`. Ein `MessageElement` besteht aus einem Bezeichner, einem Wert und dem Datentyp des Wertes. Zulässige Typen sind `String`, `byte[]` sowie `Object` für beliebige Objekte. Für jeden Datentyp existiert ein eigener Konstruktor.

Der Wert eines `MessageElement`-Objekts kann mit den Zugriffsmethoden `getStringValue`, `getByteValue` und `getObjectValue` ermittelt werden, sofern der gespeicherte Datentyp in den entsprechenden Zieltyp konvertiert werden kann. Tabelle 3.1 zeigt die zulässigen Konvertierungen.

	`String`	`byte[]`	`Object`
`String`	x	x	x
`byte[]`	x	x	
`Object`	x		x

Tabelle 3.1: Mögliche Konvertierungen der Datentypen eines MessageElements

Im Falle einer ungültigen Konvertierung wird der Wert `null` zurückgegeben. Der Name und der Datentyp eines Wertes können mit den Methoden `getName` beziehungsweise `getType` ermittelt werden.

EventMessage

Nachrichten werden in der Middleware in Form einer `EventMessage` verschickt. Eine `EventMessage` enthält dazu alle notwendigen Informationen für den Versand der Nachricht sowie mehrere `MessageElement`-Objekte in denen der Inhalt der Nachricht hinterlegt wird.

Für den Versand einer Nachricht werden die Kennungen des Dienstes und des Knotens auf dem der Dienst läuft als Quell-Informationen hinterlegt, um dem Empfänger die Herkunft der Nachricht mitzuteilen. Als Ziel einer Nachricht kann entweder ein Dienst auf einem bestimmten Knoten spezifiziert werden oder eine Kombination aus verschiedenen Broadcast-Varianten. Für die Kennung des Zielknotens steht als Broadcast-Variante `DESTINATION_NODE_BROADCAST` zur Verfügung, für den Dienst `DESTINATION_SERVICE_BROADCAST`, um alle Dienste

zu benachrichtigen und `DESTINATION_SERVICE_BY_TYPE`, um Dienste zu benachrichtigen, die sich für einen bestimmten Typ von Nachrichten angemeldet haben.

Da die Zieladresse einer Nachricht aus der Angabe des Knotens und eines Dienstes besteht, können folgende Kombinationen auftreten.

1. *Vollqualifizierte Zieladresse*
   Hierbei wird sowohl für den Zielknoten, als auch für den Dienst die entsprechende Kennung hinterlegt. Die Nachricht wird dann nur den Dienst auf dem angegebenen Knoten gesendet.

2. *Broadcast für Knoten mit spezifiziertem Dienst*
   Wird für die Zieladresse des Knotens keine Kennung, sondern `DESTINATION_NODE_BROADCAST` angegeben, verschickt die Middleware die Nachricht an alle Knoten. In Kombination mit einer spezifischen Dienstkennung entspricht diese Form der vollqualifizierten Zieladresse, da jeder Dienst über eine eindeutige Kennung verfügt und somit unter dieser Kennung nur einmal im System vorhanden ist.

3. *Broadcast für Knoten und Dienste*
   Wird für den Knoten als Kennung `DESTINATION_NODE_BROADCAST` und für den Dienst die Kennung `DESTINATION_SERVICE_BROADCAST` angegeben, wird die Nachricht an alle Knoten der Middleware und an alle Dienste gesendet.

4. *Broadcast für Knoten mit typisierter Auslieferung*
   Ein häufiger Fall bei der Nachrichtenvermittlung ist der Versand von Informationen, die an alle Dienste im System gesendet werden sollen, die sich für den Typ einer Nachricht beim `EventDispatcher` registriert haben. Dazu wird beim Knoten die Broadcast-Kennung hinterlegt und beim Dienst die Kennung `DESTINATION_SERVICE_BY_TYPE`. Der `EventDispatcher` entscheidet dann anhand des Typs der Nachricht, an welche Dienste diese Nachricht ausgeliefert wird.

5. *Broadcast für Dienste auf einem Knoten*
   In diesem Fall ist eine eindeutige Knotenkennung angegeben, für den Dienst jedoch die Kennung `DESTINATION_SERVICE_BROADCAST` hinterlegt. Damit wird die Nachricht an den angegebenen Knoten gesendet und dort an alle vorhandenen Dienste.

6. *Typisierte Auslieferung auf einem Knoten*
   Bei der typisierten Auslieferung auf einem Knoten wird die Nachricht an den angegebenen Knoten versendet und der `EventDispatcher` ermittelt alle Dienste, die sich für den angegebenen Typ dieser Nachricht registriert haben.

Obwohl insgesamt sechs verschiedene Möglichkeiten bestehen, eine Nachricht zu versenden, sind nur drei der Kombinationen tatsächlich relevant. Die Varianten 1 (vollqualifizierte Zieladresse) wird beispielsweise bei der Beantwortung eines Requests benutzt. Die Variante 4 (Broadcast für Knoten mit typisierter Auslieferung) findet Anwendung bei der Verbreitung von Sensordaten, Events oder Anfragen (Requests) mit unbekannten Empfängern. Die Variante 6 (Typisierte Auslieferung auf einem Knoten) wird benutzt, wenn der Knoten bekannt ist, auf dem sich potentielle Interessenten der Nachricht befinden. Die Alternativen 2, 3 und 5 finden eher in Einzelfällen Anwendung.

Neben dem Typ, der Quell- und Zieladresse besitzt eine Nachricht einen Modus, der dem Empfänger einen Hinweis auf die Verarbeitung der Nachricht gibt. Eine Nachricht kann mit einem von drei Modi klassifiziert werden.

1. *Event*
   Nachrichten mit dem Modus Event werden von Diensten verschickt, die andere Dienste über einen bestimmten Sachverhalt informieren möchten, aber keine Reaktion erwarten. Ein typisches Beispiel einer Event-Nachricht ist die Verbreitung oder Weitergabe von Sensor- oder Statusinformationen an andere Dienste.

2. *Request*
   Ein Request wird verschickt, wenn der Sender der Nachricht eine Antwort auf seine Anfrage erwartet. Der Request ist somit das Äquivalent zu einem Methodenaufruf mit dem Unterschied, dass bei der asynchronen Nachrichtenvermittlung keine Blockierung des Senders stattfindet.

3. *Response*
   Eine Antwort (Response) wird von einem Dienst verschickt, der zuvor eine Anfrage in Form eines Requests empfangen hat. In der Response ist es möglich auch mehrere Werte in der Antwort zu verschicken. Bei entfernten Methodenaufrufen ist dies nur möglich durch Übergabe komplexer Ergebnisobjekte oder von Objektreferenzen in der Parameterliste.

Die Ziel- und Quelladressen für den Knoten und den Dienst können mit den Methoden `getSourceNodeID`, `getSourceServiceID`, `getDestinationNodeID` und `getDestinationServiceID` ermittelt werden. Der Typ einer Nachricht kann mit der Methode `getType` und der Modus mit `getMode` erfragt werden.

EventMessageFactory

Die `EventMessageFactory` wird zur Erzeugung einer `EventMessage` benutzt. Sie bietet dazu eine Reihe verschiedener Methoden an, mit denen die notwendigen

Informationen zum Versand der Nachricht auf einfache Art und Weise übergeben werden können.

So können beispielsweise mit der Methode `createBraodcastEventMessage` unter Angabe eines Dienstes als Quelle der Nachricht und eines Nachrichtentyps, Broadcast-Nachrichten erstellt werden. Für das Erzeugen einer Antwort (Response) auf eine Anfrage (Request) kann die Methode `createResponseEventMessage` benutzt werden. Es müssen lediglich der Empfängerdienst, die ursprüngliche Nachricht und der Nachrichtentyp der Antwort übergeben werden. Die `EventMessageFactory` kann daraus alle notwendigen Informationen extrahieren und ein `EventMessage`-Objekt erstellen, dem dann `MessageElement`-Objekte hinzugefügt werden können.

EventDispatcher

Der `EventDispatcher` übernimmt in der Middleware die zentrale Rolle der lokalen Nachrichtenverteilung. Ein Dienst, der eine Nachricht versenden möchte, ruft beim `EventDispatcher` die Methode `sendMessage` auf. Eine Nachricht kann entweder aufgrund einer eindeutigen Dienstidentifikation oder anhand des Nachrichtentyps an einen Dienst zugestellt werden. Nachrichten beinhalten einen Typ der Form:

```
type.subtype.subsubtype
```

Die einzelnen Subtypen in der Typstruktur sind jeweils durch einen Punkt voneinander getrennt. Ein Beispiel könnte lauten:

```
SmartDoorplate.Locationtracking.LocationChangeMessage
```

Der Typ der Nachricht gibt in diesem Fall an, dass es sich um eine Nachricht der SmartDoorplate-Applikation handelt und dass das Locationtracking eine Nachricht über die Veränderung der Position eines Benutzers bekannt gibt. Die neue Position und die Kennung des Benutzers sind wiederum in den Elementen der Nachricht (`MessageElement`) hinterlegt.

Mit dieser Vorgehensweise können beliebige Hierarchien von Nachrichtentypen aufgebaut werden. Sinnvollerweise wird mindestens für die Dienste, die an einer Applikation mitwirken, eine Taxonomie erstellt, mit der die Nachrichten in entsprechende Klassen und Unterklassen kategorisiert werden. Denkbar wäre auch eine grobe Klassifizierung für einen größeren Zusammenhang zu definieren, der beispielsweise Sensoren entsprechend klassifiziert, um eine applikationsunabhängige Klassifizierung anzubieten.

Dienste können sich beim `EventDispatcher` mit der Methode `addService` anmelden, um mitzuteilen, dass sie beim Eintreffen von Nachrichten eines bestimmten Typs benachrichtigt werden wollen. Dabei ist es nicht zwingend notwendig einen vollqualifizierten Typ anzugeben. Dienste können sich auch für übergeordnete Typen anmelden. Alle Nachrichten vom angegebenen Typ, oder Untertypen davon, werden an die angemeldeten Dienste geleitet.

Dienste können somit aufgrund mehrerer Möglichkeiten Empfänger einer Nachricht sein.

1. *Broadcast*
   Eine Nachricht mit der Zielkennung `DESTINATION_SERVICE_BROADCAST` wird an alle Dienste des Knotens geleitet, unabhängig vom Typ der Nachricht.

2. *Typisierte Nachricht*
   Eine Nachricht mit der Zielkennung `DESTINATION_SERVICE_BY_TYPE` wird an alle Dienste verteilt, die sich für den in der Nachricht spezifizierten Typ oder einen Obertyp angemeldet haben.

3. *Eindeutige Zielkennung*
   Enthält eine Nachricht keine der beiden zuvor genannten Zielkennungen, wird davon ausgegangen, dass die Zielkennung einer Dienstkennung entspricht. Der `EventDispatcher` versucht die Nachricht an den entsprechenden Dienst zu übergeben.

Die typisierte Nachrichtenvermittlung der Middleware beruht auf der Auswertung der Nachrichtentypen durch den `EventDispatcher`. Bei einer eingehenden Nachricht wird zunächst die Zielkennung geprüft. Ist diese identisch mit `DESTINATION_SERVICE_BY_TYPE` wird der Typ der Nachricht ermittelt, anhand dessen alle Dienste benachrichtigt werden, die sich für den vollqualifizierten Typ angemeldet haben. Anschließend wird der letzte Untertyp entfernt und es werden wieder alle Dienste ermittelt, die sich für diesen Nachrichtentyp angemeldet haben. Dieses Verfahren wird so lange angewendet, bis auch der Haupttyp der Nachricht geprüft wurde.

Für das obige Beispiel würde der `EventDispatcher` versuchen, Dienste zu finden, die sich für einen der folgenden drei Nachrichtentypen angemeldet haben:

```
SmartDoorplate.Locationtracking.LocationChangeMessage
SmartDoorplate.Locationtracking
SmartDoorplate
```

Mit der Methode `removeService` kann ein Dienst wieder abgemeldet werden und empfängt dann auch keine Nachrichten mehr. Die Abmeldung kann entweder nur

auf einen bestimmen Nachrichtentyp beschränkt sein, oder den Dienst vollständig abmelden.

Der `EventDispatcher` entscheidet anhand der Knotenzieladresse einer Nachricht, ob die Nachricht lokal zugestellt werden kann, oder ob die Nachricht über einen TransportConnector an einen anderen Knoten verschickt werden muss. Da es möglich ist, mehrere TransportConnectoren in der Middleware gleichzeitig zu benutzen, muss der `EventDispatcher` entscheiden, mittels welchem TransportConnector eine Nachricht an den gewünschten Knoten zugestellt werden kann. Zu diesem Zweck führt der `EventDispatcher` eine Liste aller angemeldeten TransportConnectoren. Wenn eine Nachricht von einem entfernten Knoten ankommt, wird dem `EventDispatcher` mit der Nachricht auch die Kennung des TransportConnectors mitgeteilt, über den diese Nachricht empfangen wurde. Mit dieser Information kann der `EventDispatcher` beim Versenden einer Nachricht entscheiden, über welchen TransportConnector die Nachricht verschickt werden muss. Kann ein Knoten über mehrere TransportConnectoren erreicht werden, wird einer der verfügbaren TransportConnector für den Versand der Nachricht ausgewählt.

Wurde für einen Zielknoten noch kein zugehöriger TransportConnector identifiziert, wird zunächst geprüft, über welchen TransportConnector der gesuchte Knoten erreicht werden kann. Dieser Fall tritt jedoch nur dann ein, wenn der Sendeknoten nach dem Zielknoten gestartet wurde und beide noch nicht miteinander kommuniziert haben, weil dann der Sendeknoten das Advertisement des Zielknotens nicht erhalten hat.

3.3.4 Monitore der Vermittlungs- und Transportschicht

Die Umsetzung der Selbst-X-Eigenschaften erfordern ein hohes Maß an Wissen über die Vorgänge in der Middleware. Diese Informationen können nur gewonnen werden, wenn die Abläufe ständig beobachtet werden. Dafür ist es notwendig, an unterschiedlichen Stellen in der Middleware Kontrollpunkte vorzusehen, bei denen Monitore zur Beobachtung der Abläufe eingebracht werden können.

Monitore dienen primär der Beobachtung des Systems. Sie sind aber auch in der Lage Informationen in den Kommunikationsfluss einzubringen. Diese Vorgehensweise hat den enormen Vorteil, dass die Monitore über diesen Weg keine eigene aktive Kommunikation benötigen und somit Kommunikationsbandbreite eingespart werden kann. Diese passive Form der Kommunikation zwischen den Monitoren ist eines der besonderen Merkmale der Middleware, ebenso wie die Forderung, dass das Monitoring ausschließlich lokal erfolgt. Die Monitore sind in der Lage, ihre Informationen in den vorhandenen Kommunikationsfluss auf der einen Seite aufzuprägen und auf der anderen Seite wieder zu extrahieren.

In Abbildung 3.11 ist das Klassendiagramm der Monitore dargestellt. Das Interface `MessageMonitor` ist die Oberklasse, von dem alle Monitore abgelei-

tet sind. Für die Kommunikation wird zwischen den beiden Richtungen für eingehende und ausgehende Nachrichten unterschieden und durch entsprechende Interfaces (`OutgoingMonitor` und `IncomingMonitor`) deklariert. Weiterhin wird zwischen Monitoren auf der Transport- und der Dienst-Ebene unterschieden. Damit ergeben sich die vier Monitortypen `OutgoingTransportMonitor`, `IncomingTransportMonitor`, `OutgoingMessageMonitor` und `IncomingMessageMonitor`. Neben den Monitoren, die in den Kommunikationsweg eingebunden werden können, gibt es noch Monitore vom Typ `SystemMonitor`, die allgemeine Informationen über das System, beziehungsweise den Knoten sammeln können und nicht direkt vom Kommunikationsfluss betroffen sind. Die Unterscheidung in die verschiedenen Monitor-Typen dient der späteren Zuordnung zu den vorhandenen `MonitorQueues`.

Eine `MonitorQueue` umfasst die Funktionalität zur Organisation von Monitoren eines bestimmten Typs. Im Konstruktor einer `MonitorQueue` wird der Typ der zu verwaltenden Monitore angegeben. In eine `MonitorQueue` können dann nur Monitore vom entsprechenden Typ hinzugefügt werden, um eine fälschliche Zuordnung der Monitore zu vermeiden. Monitore können mit der Methode `addMessageMonitor` einer `MonitorQueue` hinzugefügt werden. Dabei kann der Monitor entweder am Ende oder an einer bestimmten Stelle hinzugefügt werden. Wird kein weiterer Parameter angegeben, wird der Monitor an das Ende angefügt. Wird ein Index als Parameter angegeben, wird der Monitor an der entsprechenden Stelle in der Monitorliste eingefügt. Es kann auch ein Monitor als Referenz angegeben werden, hinter dem der neue Monitor eingefügt werden soll. Auf diese Weise lassen sich reproduzierbare Verarbeitungsreihenfolgen aufbauen.

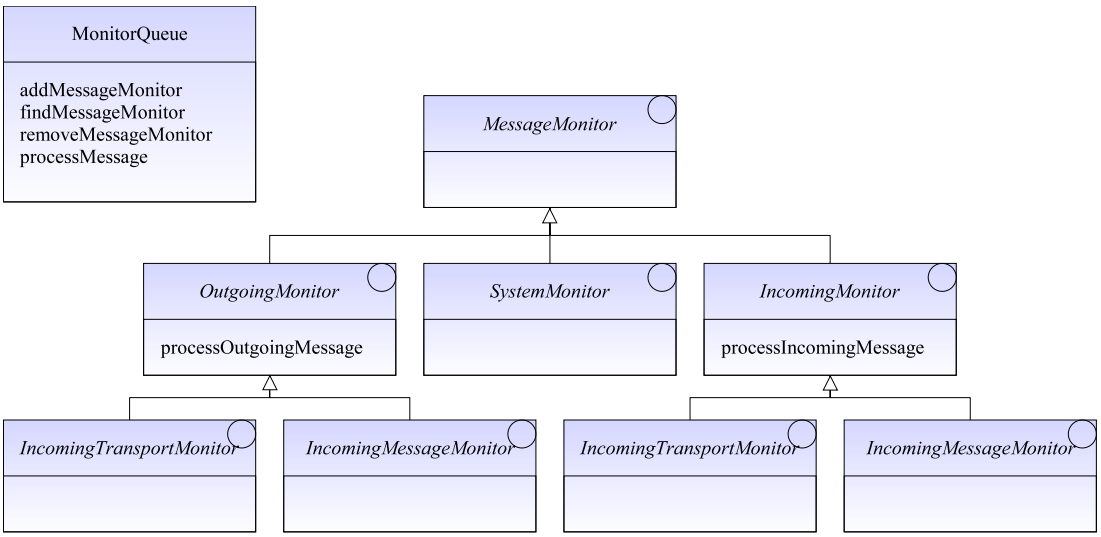

Abbildung 3.11: Klassendiagramm der Monitore

Monitore können zu jedem Zeitpunkt aus einer `MonitorQueue` entfernt werden, indem die Methode `removeMessageMonitor` aufgerufen wird. Es kann entweder der Monitor selbst oder eine Position angegeben werden an der ein Monitor entfernt werden soll. Mit der Methode `findMessageMonitor` kann die Position eines Monitors in einer `MonitorQueue` ausfindig gemacht werden.

Wie bereits in Abschnitt 3.1.2 erläutert wurde, wird an verschiedenen Stellen während dem Nachrichtenversand eine Nachricht durch entsprechende `MonitorQueues` geleitet. Der `EventDispatcher` und der `TransportConnector` besitzen jeweils eine `MonitorQueue` für eingehende und eine für die ausgehenden Nachrichten. Soll eine Nachricht von einer `MonitorQueue` verarbeitet werden, wird die Nachricht der Methode `processMessage` der `MonitorQueue` übergeben. Die `MonitorQueue` übergibt dann die Nachricht an jeden Monitor in der Reihenfolge, wie sie durch das Hinzufügen der Monitore definiert wurd.

3.3.5 Diensteschicht

Ubiquitäre Systeme zeichnen sich durch eine Vielzahl verteilter Geräte und Sensoren aus, deren Zusammenwirken dem Anwender den Eindruck einer allgegenwärtigen und überall verfügbaren Applikation vermitteln soll. Heutige Anwendungen unterscheiden sich dadurch, dass sie meist explizit auf einem Computer aufgerufen werden müssen und nur in Ausnahmefällen auf einer verteilten Architektur beruhen. Ein Beispiel für eine Architektur, die auf verteilten Diensten basiert, ist die Web Services Activity [81], die auf der Webservice Description Language (WSDL) [80] und auf SOAP [82] beruht.

Für die Entwicklung von Anwendungen auf Basis der Organic Ubiquitous Middleware wird ein Umdenken gefordert. Applikationen werden nicht mehr als monolithische Systeme implementiert, sondern entstehen durch Komposition verschiedener Dienste. Dabei steht die Forderung nach Wiederverwendbarkeit der Dienste bei der Aufteilung der Anwendungsfunktionen auf einzelne Dienste an erster Stelle.

Die Organic Ubiquitous Middleware erfüllt die Forderungen an eine verteilte und transparente Infrastruktur zur Realisierung dienstbasierter Anwendungen. Dem Entwickler werden Schnittstellen bereitgestellt, mit denen sowohl frei verteilbare, als auch an einen Knoten gebundene Dienste implementiert werden können. Abbildung 3.12 zeigt das Klassendiagramm der Diensteschicht.

Gebundene Dienste

Die Klasse `Service` dient als Basis aller Dienste in der Middleware. Jeder Dienst wird von dieser Klasse abgeleitet und erbt die Methoden dieser Klasse, die unter anderem zur Verwaltung der Dienste in der Middleware benötigt wird. Ein Dienst

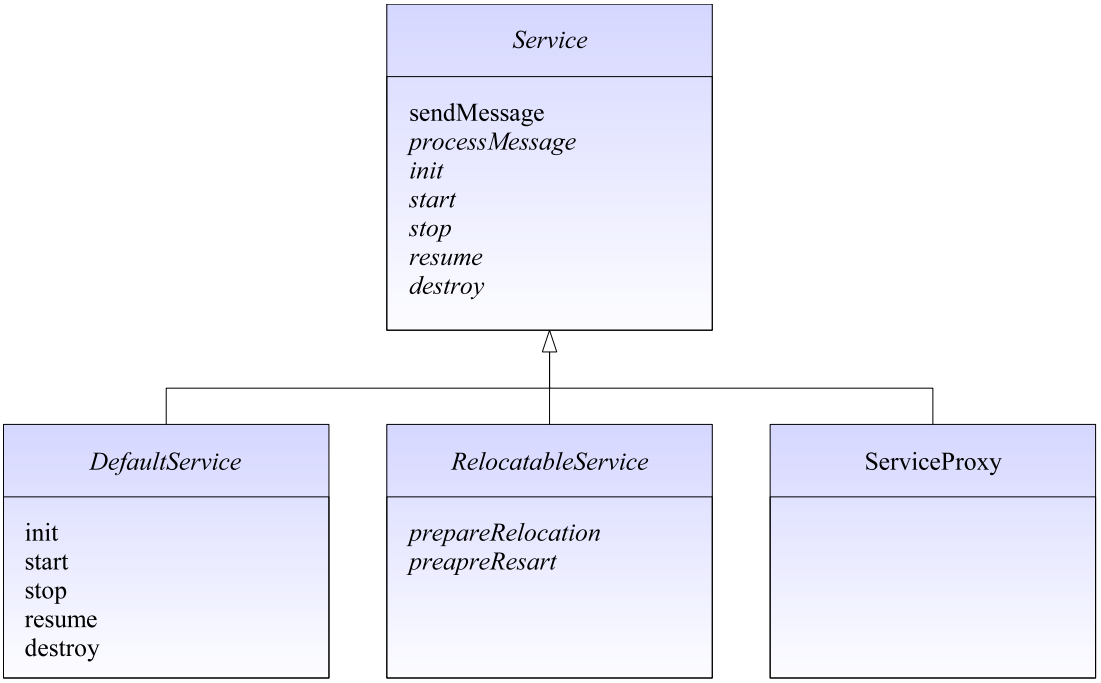

Abbildung 3.12: Klassendiagramm der Diensteschicht

muss vier Grundinformationen zu dessen Verwaltung bereitstellen. Eine eindeutige Dienstkennung zur Identifikation des Dienstes, einen Namen (nicht zwingend eindeutig), einen `ServiceType` zur Klassifizierung des Dienstes und eine Anzahl von `ServiceClasses` zur weiteren Einordnung des Dienstes in eine Taxonomie.

Die `ServiceClasses` werden in der aktuellen Version der Middleware nicht benutzt, sollen später jedoch beim Discovery von Diensten herangezogen werden. Über diesen Weg sollen Dienste nicht anhand ihrer angebotenen Schnittstellen oder Methoden gefunden werden, sondern anhand ihrer funktionalen Zuordnung. Aus diesem Grund kann ein Dienst auch zu mehreren `ServiceClasses` gehören, da er möglicherweise mehrere Funktionen aus unterschiedlichen Bereichen der Funktionstaxonomie anbietet. Die Entscheidung, welche Parameter der Dienst zur Erbringung der gesuchten Funktion benötigt, soll mittels einer WSDL-Beschreibung ohne Bindung (Binding) an eine konkrete Implementierung erfolgen. Diese Erweiterung im Zusammenhang mit der typisierten Nachrichtenvermittlung der Organic Ubiquitous Middleware bietet einen zusätzlichen Vorteil gegenüber dem Stubs/Skeleton-Prinzip herkömmlicher Middlewaresysteme, da Dienste nicht nur anhand der implementierten Schnittstelle, sondern auch anhand funktionaler Zuordnungen gesucht werden können, ohne dass eine exakte Übereinstimmung der Schnittstellen erforderlich ist.

Um einen neuen Dienst zu implementieren genügt es, die abstrakten Methoden `init`, `start`, `stop`, `resume`, `destroy`, sowie `processMessage` zu implementieren. Die Methode `init` wird aufgerufen, nachdem eine Instanz eines Dienstes erstellt wurde. In dieser Methode kann der Dienst notwendige Initialisierungen vornehmen. Direkt bevor der Dienst gestartet wird, wird die Methode `start` aufgerufen. In dieser Methode kann sich der Dienst beispielsweise beim `EventDispatcher` registrieren, damit er von anderen Diensten durch ein Discovery gefunden werden kann, oder beim `EventDispatcher` für bestimmte Nachrichtentypen als Empfänger eintragen. Soll ein Dienst gestoppt werden, wird die Methode `stop` aufgerufen. Der Dienst sollte sich dann beim `EventDispatcher` austragen, damit er nicht weiterhin über eingehende Nachrichten informiert wird. Die Methode `resume` wird aufgerufen, wenn ein Dienst wieder ausgeführt werden soll, nachdem er mit `stop` zuvor angehalten wurde. Die Methode `destroy` wird aufgerufen, bevor der Dienst endgültig entfernt wird.

Jeder Dienst der Middleware wird als eigener Thread ausgeführt und verfügt über eine eigene Warteschlange für eingehende Nachrichten. Die Verarbeitung der Nachrichten erfolgt in der Methode `processMessage`, die aufgerufen wird, wenn eine neue Nachricht in der Warteschlange eingetroffen ist. Solange keine neue Nachricht in der Warteschlange vorliegt, wird der Dienst blockiert. Soll ein Dienst nicht blockiert, sondern in regelmäßigen Intervallen ausgeführt werden, während er auf eingehende Nachrichten wartet, kann eine maximale Wartezeit festgelegt werden. Ist eine maximale Wartezeit (Timeout) angegeben, entspricht diese der Zeit, die längstens gewartet wird, bis eine neue Nachricht eintrifft. Trifft innerhalb dieser Zeit keine neue Nachricht ein, wird die Methode `processMessage` trotzdem aufgerufen. In diesem Fall wird jedoch als Parameter NULL übergeben, um anzuzeigen, dass keine neue Nachricht vorhanden ist. Bei der Initialisierung eines Dienstes ist der Timeout auf 0 gesetzt, was bedeutet, dass kein Timeout benutzt werden soll und der Dienst blockiert wird. Wird ein Wert größer als 0 gesetzt, wird der Timeout als maximale Wartezeit beim Blockieren des Dienstes berücksichtigt.

Damit ein Dienst im System durch eine Discovery-Anfrage gefunden werden kann, muss er registriert sein. Um Nachrichten empfangen zu können, muss der Dienst beim `EventDispatcher` angemeldet sein. Der Unterschied zwischen Registrierung und Anmeldung besteht darin, dass bei einer Registrierung der Dienst für andere Dienste sichtbar gemacht wird, indem sein `ServiceAdvertisement` verbreitet wird. Bei der Anmeldung am `EventDispatcher` bekundet der Dienst sein Interesse an bestimmten Nachrichten. Ein Dienst kann somit entscheiden, ob er sich durch eine Registrierung im System bekanntmachen möchte oder nicht. Um Nachrichten empfangen zu können muss sich ein Dienst auf jeden Fall beim `EventDispatcher` für bestimmte Nachrichtentypen anmelden. Eine Registrierung ohne Anmeldung beim `EventDispatcher` ist folglich nur dann sinnvoll, wenn der

Dienst ausschließlich direkt benachrichtigt werden soll, da der Dienst zwar gefunden werden kann, aber keine typisierten Nachrichten empfangen kann.

Bei der Anmeldung übergibt ein Dienst dem `EventDispatcher` die Typen der Nachrichten, die er erhalten möchte. Ein Dienst kann sich für mehrere Nachrichtentypen anmelden und auch einzelne davon wieder abmelden. Wie in Kapitel 3.3.3 beim `EventDispatcher` beschrieben, werden einem angemeldeten Dienst alle Nachrichten zugestellt, deren Typ den angemeldeten oder Untertypen davon entsprechen.

Für die Registrierung und Deregistrierung eines Dienstes bietet die Klasse `Service` die Methoden `register` und `unregister`. Die Methode `register` erzeugt aus den Daten des Dienstes ein entsprechendes `ServiceAdvertisement` und übergibt dieses dem `EventDispatcher`. Der `EventDispatcher` sorgt dafür, dass das `ServiceAdvertisement` im System bekannt gemacht wird. Ein `ServiceAdvertisement` beinhaltet die Kennung, den Namen sowie den Typ des Dienstes, die Kennung des Knotens auf dem der Dienst zu finden ist und einen Array mit weiteren `ServiceClasses`.

Die Klasse `DefaultService` stellt eine Implementierung der Klasse `Service` bereit, bei der die Methoden `init`, `start`, `stop`, `resume`, `destroy` mit leeren Methodenrümpfen implementiert sind. Die Methode `processMessage` muss von einem Dienst weiterhin implementiert werden.

Bei einem Dienst, der von der Klasse `Service` oder `DefaultService` abgeleitet wird handelt es sich um einen gebundenen Dienst. Dies bedeutet, dass der Dienst nur auf dem Knoten, auf dem er gestartet wurde ausgeführt wird und nicht durch die Middleware auf einen anderen Knoten verlegt werden darf. Gebundene Dienste werden benutzt, wenn ein Dienst zum Betrieb bestimmte Hard- oder Softwarevoraussetzungen benötigt. Ein Dienst, der beispielsweise einen Sensor oder Aktuator in die Middleware integriert, muss zwangsläufig auf dem Knoten laufen, an dessen Rechner die entsprechende Hardware angeschlossen ist. Für diesen Dienst wäre eine Verlegung zwecklos, weil er seine Aufgabe nicht mehr erfüllen könnte. Ein anderes Beispiel sind Dienste, die eine Datenbank in das System integrieren. Der Aufwand, eine Datenbank auf einen anderen Knoten zu verlagern ist so hoch, dass die Datenbank immer auf dem gleichen Computer ausgeführt werden sollte. Ein Dienst, der eine Datenbank anbindet, sollte somit ebenfalls auf diesem Rechner laufen, um möglichst schnell auf Anfrageergebnisse zugreifen zu können und den Kommunikationsaufwand über das Netzwerk zu minimieren.

Verschiebbare Dienste

Verschiebbare Dienste bilden die Grundlage der Selbstkonfigurations- und Selbstoptimierungseigenschaften der Organic Ubiquitous Middleware. Da Dienste je-

doch meist über einen internen Zustand verfügen, muss bei der Verlegung verschiebbarer Dienste darauf geachtet werden, dass auch der interne Zustand nach der Verlegung wieder verfügbar ist. Es genügt somit nicht, nur die Klassen des Dienstes auf dem neuen Knoten zur Verfügung zu stellen, sondern auch den Zustand des Dienstes. Da die Middleware nicht entscheiden kann, welche Daten des Dienstes bei der Verlegung erhalten bleiben sollen, muss jeder Dienst diese Entscheidung selbst treffen.

Die abstrakte Klasse `RelocatableService` erweitert die Klasse `Service` um die Methoden `prepareRelocation` und `prepareRestart`. Die Methode `prepareRelocation` wird aufgerufen, um dem Dienst mitzuteilen, dass er auf einen anderen Knoten übertragen werden soll und seinen internen Zustand für die Übertragung vorbereiten muss. Dazu wird der Methode ein `OutputStream` als Parameter übergeben, in den der Dienst seinen internen Zustand speichern kann. Möchte der Dienst Objekte speichern, kann er mit dem `OutputStream` einen `ObjectOutputStream` erstellen und dann mit der Methode `writeObject` in den Datenstrom serialisieren.

Nachdem der Dienst auf den neuen Knoten verlegt und eine neue Instanz der Klasse erzeugt wurde, wird die Methode `prepareRestart` aufgerufen. Der Methode wird ein `InputStream` übergeben, aus dem der Dienst seine benötigten Daten wieder auslesen kann. Anschließend wird der Dienst durch den `Configurator` gestartet.

Service Proxy

Die Verlegung von Diensten gehört zu den grundlegenden Funktionen der Middleware zur Realisierung der Selbstkonfiguration und Selbstoptimierung. Das Verschieben eines Dienstes auf einen anderen Knoten bedeutet jedoch auch, dass die bisherige Information der anderen Knoten über den Aufenthaltsort des Dienstes veraltet ist. Um nicht jeden Knoten über die Verlegung eines Dienstes informieren zu müssen und damit die Middleware vor einer Überflutung mit Nachrichten zu bewahren, wird ein einfacher Mechanismus angewandt, um eine aktive Benachrichtigung zu vermeiden.

Wird ein Dienst auf einen anderen Knoten verlegt, ist er auf dem alten Knoten nicht mehr erreichbar und Anfragen an diesen Dienst würden nicht zugestellt werden können. Besonders in dem Zeitraum der Verlagerung des Dienstes gäbe es keine Möglichkeit, eine eingehende Nachricht zustellen zu können, selbst nicht an den neuen Knoten, da der Dienst möglicherweise noch nicht für den Empfang von Nachrichten bereit ist. Um diesem Umstand zu begegnen, wird ein `ServiceProxy` für den verlegten Dienst auf dem alten Knoten erzeugt. Der `ServiceProxy` agiert als Stellvertreter des Dienstes und nimmt eingehende Nachrichten entgegen, um sie an den neuen Knoten weiterzuleiten. Der `ServiceProxy` wird erzeugt, direkt

nachdem der Dienst beim `EventDispatcher` abgemeldet wurde und somit keine Nachrichten mehr empfangen kann. An dessen Stelle wird der `ServiceProxy` beim `EventDispatcher` angemeldet. Die Anmeldung ist bezüglich der Zustellung von Nachrichten eine untrennbare Aktion und kann nicht durch eine eingehende Nachricht unterbrochen werden. Damit ist sichergestellt, dass keine Nachricht durch das Verlegen eines Dienstes verloren geht.

Grundsätzlich leitet ein `ServiceProxy` alle eingehenden Nachrichten an den Dienst weiter für den er als Stellvertreter eingesetzt wurde. Der `ServiceProxy` wartet jedoch so lange mit der Weiterleitung eingehender Nachrichten, bis sich der verlagerte Dienst von dem neuen Knoten beim `ServiceProxy` gemeldet hat und damit seine Empfangsbereitschaft signalisiert. Der `ServiceProxy` speichert eingehende Nachrichten für den Zeitraum der Verlagerung eines Dienstes. Anschließend sendet er alle eingegangenen Nachrichten an den Dienst und leitet alle folgenden Nachrichten direkt weiter, ohne diese zwischenzuspeichern.

Der Ablauf einer Kommunikation stellt sich damit wie folgt dar: Ein Dienst A auf dem Knoten k_1 möchte eine Nachricht an den Dienst B schicken. Der Dienst B wurde gerade vom Knoten k_2 auf den Knoten k_3 verlagert. Der Dienst A schickt seine Nachricht an den Knoten k_2, da er über die Verlegung des Dienstes noch nicht informiert wurde. Auf dem Knoten k_2 wurde für den Dienst B ein `ServiceProxy` eingerichtet, der die eingehende Nachricht des Dienstes A entgegennimmt. Er fügt der Nachricht eine Kennung hinzu, dass er diese Nachricht weitergeleitet hat, und schickt diese an den Knoten k_3. Der Dienst B verarbeitet die Nachricht und schickt die Antwort an Dienst A zurück, ebenfalls mit einer Kennung, dass Dienst B jetzt auf dem Knoten k_3 zu finden ist. Diese Information wird auf dem Knoten k_1 dem DiscoveryService durch einen Monitor mitgeteilt und die Datenbasis bezüglich der Dienste wird aktualisiert. Ab sofort wird jede weitere Anfrage an den Dienst B direkt an den Knoten k_3 geleitet.

Durch diesen Mechanismus ist es möglich Dienste zu verschieben, ohne alle anderen Knoten über die Verlegung eines Dienstes informieren zu müssen. Der Aufwand für den zusätzlichen Versand der Nachricht vom `ServiceProxy` zum Zielknoten ist in jedem Fall geringer, als alle Knoten zu informieren. Im ungünstigsten Fall wird für jeden Knoten einmal eine Nachricht über den `ServiceProxy` gesendet bevor jeder Knoten über den neuen Standort des Dienstes informiert wurde. Da jedoch davon ausgegangen werden kann, dass jeder Knoten immer nur wenige bevorzugte Kommunikationspartner besitzt, wird dieser Fall sehr selten eintreten.

Ein `ServiceProxy` verfügt nur über eine begrenzte Lebensdauer und wird nach Ablauf dieser Zeit vom Knoten entfernt. Die Lebensdauer eines `ServiceProxy`-Objekts hängt von der Gültigkeitsdauer des `ServiceAdvertisement`-Objekts ab. Nach Ablauf der Gültigkeitsdauer muss das `ServiceAdvertisement` entweder erneut aktiv publiziert werden, oder es muss durch ein Discovery erneut gefunden werden, womit der neue Standort des Dienstes wieder bekannt wäre und der `ServiceProxy` nicht länger benötigt wird.

3.3.6 Basisdienste

Die Middleware besitzt zwei Dienste, die als integrale Bestandteile der Middleware die Konfiguration und das Verlegen, sowie das Finden von Diensten ermöglicht. Im Unterschied zu anderen Diensten existiert jeweils nur eine Instanz der Klasse `ConfiguratorService` und `DiscoveryService`, die jeweils als Singleton realisiert sind.

ConfiguratorService

Der `ConfiguratorService` besteht aus zwei Klassen. Dem eigentlichen `ConfiguratorService`, der Konfigurationsnachrichten empfangen und verarbeiten kann und dem `Configurator`, der für die Durchführung der Konfigurationsaufgaben verantwortlich ist.

Beim Start der Middleware wird zunächst der Knoten und anschließend mindestens ein TransportConnector gestartet, damit der Knoten in das Netzwerk eingebunden wird und mit den anderen Knoten kommunizieren kann. Anschließend wird die einzige Instanz (Singleton) der Klasse `Configurator` erstellt. Dieser wiederum erstellt eine Instanz der Klasse `ConfiguratorService` und `DiscoveryService`.

Der `Configurator` verwaltet alle Dienste und Monitore und ist für das Starten und Beenden verantwortlich. Der Ablauf einer Selbstkonfiguration wird in Kapitel 4 erläutert. Der `Configurator` besitzt darüber hinaus die Funktionalität, um eine Verlegung eines Dienstes zu initiieren, beziehungsweise einen verlegten Dienst zu reinitialisieren und zu starten, wie in Abschnitt 3.3.5 beschrieben.

Da der `Configurator` selbst keine Nachrichten empfangen kann, wird der `ConfiguratorService` benutzt, um alle Nachrichten bezüglich der Konfiguration und Verlegung von Diensten entgegen zu nehmen und gegebenenfalls an den `Configurator` weiterzuleiten. Wurde die Verlegung eines Dienstes durch den Organic Manager angestoßen, sendet der `Configurator` den Dienst und seine serialisierten Daten an den Empfängerknoten. Dort wird die Nachricht vom `ConfiguratorService` entgegengenommen und an den `Configurator` weitergeleitet, der einen `RelocatableServiceRunner` auffordert eine Instanz des Dienstes zu erstellen. Der `RelocatableServiceRunner` ist für das Laden der Klassen des Dienstes verantwortlich. Da die zugrunde liegende Kommunikation asynchron arbeitet, ist das Laden der Klassen von einem anderen Knoten ebenfalls mittels asynchroner Kommunikation zu erledigen und stellt eine Besonderheit der Middleware dar. Bisher wurde noch kein asynchrones Verfahren zum entfernten Laden von Klassen implementiert. Dieser Vorgang wird in Kapitel 3.3.8 im Detail erläutert.

Wird der `Configurator` aufgefordert einen Dienst lokal zu erstellen, versucht er zunächst eine Instanz des Dienstes zu erzeugen. Anschließend werden die Metho-

den `init` und `start` nacheinander aufgerufen, damit sich der Dienst initialisieren und alles Notwendige für den Start des Dienstes vorbereiten kann. Dann wird ein eigener Thread für die Ausführung des Dienstes erstellt und gestartet.

DiscoveryService

Die Dienste einer Applikation können sich zu unterschiedlichen Zeitpunkten auf unterschiedlichen Knoten der Middleware befinden, sofern es sich um `RelocatableServices` handelt. Um eine Anfrage gezielt an den richtigen Dienst senden zu können, muss der Knoten ausfindig gemacht werden, auf dem sich der Dienst gerade befindet. Diesem Zweck dient der `DiscoveryService`.

Die normale Arbeitsweise verteilter Discovery-Dienste besteht darin, aktiv den Aufenthaltsort eines Dienstes zu erfragen, sofern dieser nicht bereits bekannt ist. In einem System, in dem die Dienste ständig ihren Aufenthaltsort ändern können, würde diese Vorgehensweise das System mit Anfragen überfluten.

Aus diesem Grund arbeitet der `DiscoveryService` sehr eng mit dem `ConfiguratorService` zusammen. Sofern der Aufenthaltsort eines Dienstes nicht lokal festgestellt werden kann, wird im Netzwerk aktiv nach dem Dienst gesucht, wie es von anderen Discovery-Diensten bekannt ist. Dieser Fall sollte jedoch so gut wie nie eintreten, da der `DiscoveryService` eigentlich immer aktuelle Informationen lokal verfügbar haben sollte, selbst wenn ein Dienst verlegt wurde.

Der `DiscoveryService` beobachtet ständig die eingehenden Nachrichten und untersucht die Senderinformation auf neue Daten hin. Wird eine neue oder aktuellere Information gefunden aktualisiert er automatisch seine Datenbasis.

Angenommen der `DiscoveryService` des lokalen Knotens kennt die aktuellen Aufenthaltsorte der Dienste und ein Dienst würde die Position eines anderen Dienstes erfragen, könnte der `DiscoveryService` sofort antworten. Gehen wir davon aus, dass der gesuchte Dienst gerade eben auf einen anderen Knoten verlegt wurde, so wäre die Antwort des `DiscoveryService` falsch. Trotzdem schickt der Dienst seine Nachricht an den Knoten, auf dem der Empfängerdienst erwartet wird. Dort läuft anstelle des verlegten Dienstes für eine gewisse Zeit ein `ServiceProxy`, der die Anfragen an den verlegten Dienst weiterleitet und eine Information hinzufügt, dass eine Weiterleitung der Nachricht erfolgte.

Der verlegte Dienst beantwortet die Anfrage auf direktem Weg und fügt die Information des `ServiceProxy` hinzu. Damit weiß der Senderknoten der ursprünglichen Nachricht, dass der Empfängerdienst auf einen anderen Knoten verlegt wurde und der `DiscoveryService` wird entsprechend informiert. Diese Vorgehensweise verhindert zum einen, dass beim Verlegen von Diensten jedes mal alle `DiscoveryServices` benachrichtigt werden müssen und zum anderen ist damit die Middleware in der Lage für eine gewisse Zeit inkonsistente Informatio-

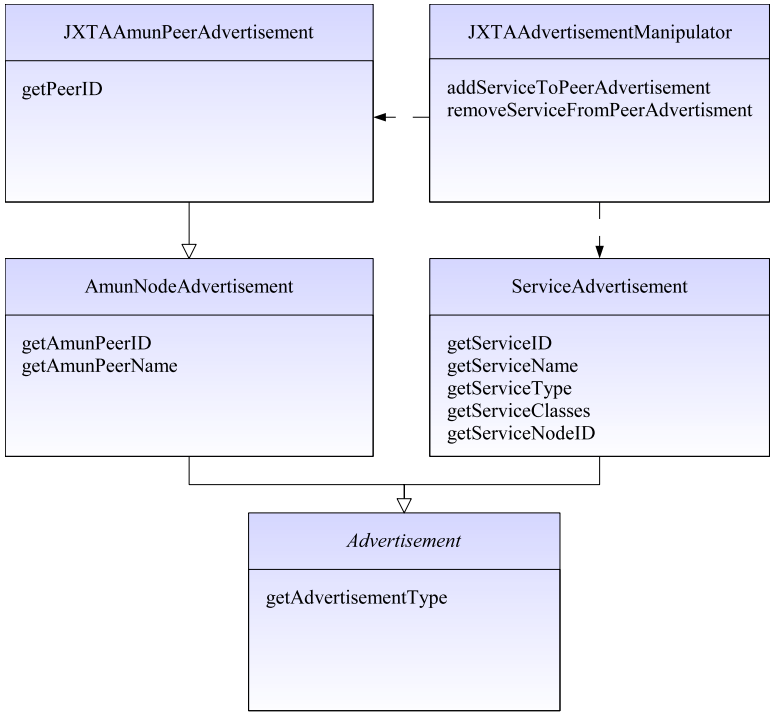

Abbildung 3.13: Advertisementarten der Middleware und des JXTATransport-Connectors

nen des `DiscoveryService` zu tolerieren, da diese durch weitere Kommunikation automatisch aufgelöst werden.

Für den Fall, dass der `ServiceProxy` des verlegten Dienstes nicht mehr erreicht werden kann, muss der Dienst aktiv gesucht (discovered) werden. Dieser Zustand tritt jedoch nur für Kommunikationspfade zwischen Diensten auf, die sehr selten miteinander kommunizieren. In diesem Fall kann der zusätzliche Aufwand für ein Discovery in Kauf genommen werden.

Die Daten, die der `DiscoveryService` zum Austausch zwischen den Knoten benutzt, um Informationen über andere Knoten und deren Dienste in Erfahrung zu bringen, werden in sogenannten `Advertisements` hinterlegt. In der Middleware gibt es ein `AmunNodeAdvertisement` und ein `ServiceAdvertisement`.

Im `ServiceAdvertisement` wird der Name, die Identifikation, der Typ, die Dienstklassen des Dienstes und die Identifikation des aktuellen Knotens hinterlegt, auf dem der Dienst läuft. Das `ServiceAdvertisement` besitzt darüber hinaus die Möglichkeit ein teilweise ausgefülltes `ServiceAdvertisement` mit seinen Informationen zu vergleichen um entscheiden zu können, ob es bei einer Suchanfrage als Ergebnis infrage kommt. Dabei können reguläre Ausdrücke in

Java [64] benutzt werden, um den gesuchten Dienst nicht vollständig qualifizieren zu müssen.

Das `AmunNodeAdvertisement` enthält die Informationen über den Knoten, den es beschreibt. Dies sind lediglich der Name des Knotens und dessen eindeutige Identifikation.

Sowohl das `ServiceAdvertisement`, als auch das `AmunNodeAdvertisement` sind von der Oberklasse `Advertisement` abgeleitet. Beide Advertisement-Arten werden, bei Verwendung des `JXTATransportConnector` in dem `JXTAAmunPeerAdvertisement` zusammengefasst, um einen Knoten vollständig zu beschreiben. Der `JXTAAdvertisementManipulator` ermöglicht die Veränderung des `JXTAAmunPeerAdvertisement` bei sich ändernden Informationen. Das `JXTAAmunPeerAdvertisement` wird im JXTA-Peer-to-Peer-Netzwerk zwischen den Knoten in Form von XML-Dokumenten ausgetauscht.

3.3.7 Der Organic Manager

Der Organic Manager ist der zweite große Teil der Middleware. Neben den Komponenten der Middleware, die für die Verwaltung von Diensten und der Zustellung von Nachrichten verantwortlich sind, beinhaltet der Organic Manager die für die Realisierung der Selbst-X-Eigenschaften notwendigen Komponenten.

Erst durch die Erweiterung der Middleware mit dem Organic Manager entsteht eine Observer-Controller-Architektur, die die Umsetzung der Selbst-X-Eigenschaften erlaubt und gleichzeitig eine klare Trennung zwischen den einzelnen Bereichen der Middleware ermöglicht. Abbildung 3.14 zeigt die Komponenten des Organic Managers.

Nachfolgend werden die Komponenten des Organic Managers erklärt. Auf die Dienste und Komponenten der Selbstkonfiguration und Selbstoptimierung wird hier nicht weiter eingegangen, da sie in den entsprechenden Kapiteln im Detail erörtert werden.

InformationPool

Voraussetzung für die Umsetzung der Selbst-X-Eigenschaften ist die kontinuierliche und intensive Beobachtung der Vorgänge in der Middleware. Die Beobachtung und Sammlung der Daten erfolgt durch Monitore, die in der Transportschicht und der Diensteschicht eingebunden werden können, sowie durch SystemMonitore, die den Systemzustand kontinuierlich beobachten.

Die Monitore sind jedoch nur zur Informationsgewinnung und nicht zur Informationsverarbeitung vorgesehen. Die gewonnenen Daten können von unterschiedlichen Komponenten weiter verarbeitet werden. Beispielsweise können Dienste an

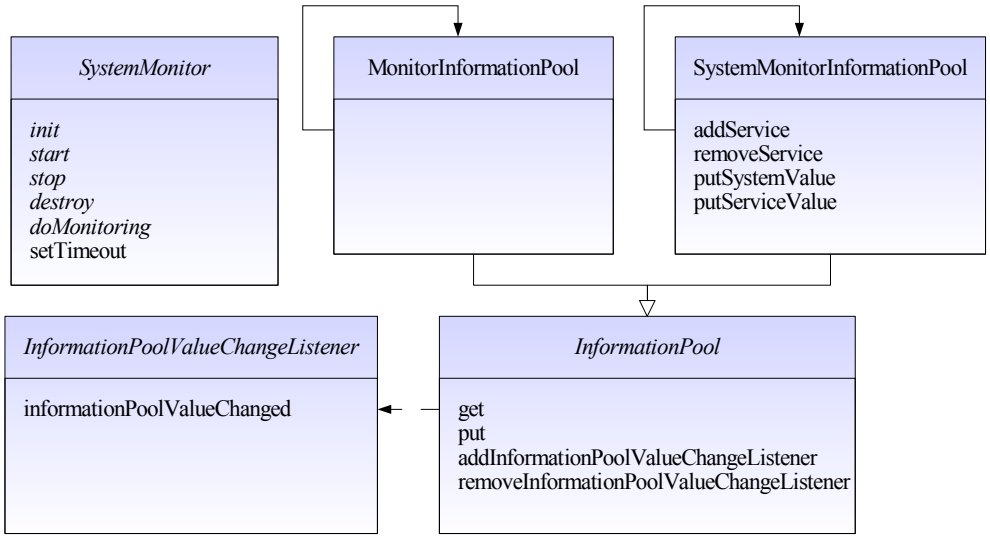

Abbildung 3.14: Die Klassen des Organic Managers, die für die Verwaltung der gewonnenen Daten zuständig sind.

den Informationen interessiert sein, oder eine Metrik zur Berechnung der Lasten auf einem Knoten könnte die Daten benötigen. Damit ein Monitor nicht für die Auslieferung der Daten an die entsprechenden Komponenten verantwortlich ist und die Komponenten, die an den Informationen interessiert sind, nicht von einem bestimmten Monitor abhängig sind, gibt es den `MonitorInformationPool`.

Die abstrakte Klasse `InformationPool` stellt eine aktive Informationsquelle dar und implementiert die grundlegenden Mechanismen, um Informationen als Parameter-Wert-Paare hinterlegen zu können. Interessenten müssen die Schnittstelle `InformationPoolValueChangeListener` implementieren und können sich für bestimmte Daten beim `InformationPool` registrieren. Übergibt ein Monitor neue Daten an den `InformationPool` wird zusätzlich zu dem Wert ein Schlüssel übergeben. Der `InformationPool` speichert den übergebenen Wert und sucht nach Komponenten, die sich für den übergebenen Schlüssel registriert haben. Werden entsprechende Komponenten gefunden, wird die Methode `informationPoolValueChanged` aufgerufen und der aktuelle Wert übergeben.

Der Publisher/Subscriber-Mechanismus verhindert, dass Interessenten ständig neue Informationen erfragen müssen. Außerdem können sich Komponenten für mehrere Schlüssel registrieren und werden immer dann benachrichtigt, wenn eine Information zu einem der Schlüssel eingetragen wurde.

Der `InformationPool` kann auch als Austauschplattform für die Komponenten des Organic Managers benutzt werden. Komponenten können sich somit gegenseitig Informationen zukommen lassen ohne dass sich der Erzeuger und der Verbraucher der Information kennen müssen. Somit ist nicht nur gewährleistet, dass

die Komponenten des Organic Managers, die für die Realisierung der Selbst-X-Eigenschaften zuständig sind, unabhängig implementiert werden können, sondern auch, dass erzeugte Informationen durch diesen Eventmechanismus immer sofort zur Weiterverarbeitung bereitstehen und die interessierten Komponenten informiert werden.

MonitorInformationPool

Von der Klasse `InformationPool` ist die Klasse `MonitorInformationPool` abgeleitet, die für die Bereitstellung und Verbreitung der Daten aus den Monitoren verantwortlich ist. Der `MonitorInformationPool` ist als Singleton implementiert, sodass es nur eine Instanz dieser Klasse gibt und jeder Monitor seine Informationen dort ablegen und jeder Dienst sich registrieren kann.

SystemMonitorInformationPool

Der `SystemMonitorInformationPool` ist ebenfalls von der Klasse `InformationPool` abgeleitet und erbt alle Methoden, die für die Verwaltung der Informationen bereitgestellt werden. Er erweitert die Klasse um Funktionen zur Unterscheidung zwischen Werten, die sich auf das Gesamtsystem beziehen und Werten, die sich auf einzelne Dienste beziehen.

Hierzu besteht die Möglichkeit dem `SystemMonitorInformationPool` die Kennung eines Dienstes zu übergeben, für den anschließend verschiedene Systemparameter hinterlegt werden, wie beispielsweise die prozentuale Last des Prozessors durch den Dienst.

SystemMonitor

Die abstrakte Klasse `SystemMonitor` ist die Basis für die Implementierung spezifischer Systemmonitore mit denen Informationen über das System und über die Dienste gesammelt werden können. Im Vergleich zu den anderen Monitoren, die in den Kommunikationsfluss beim Versenden und Empfangen von Nachrichten integriert sind, ist der Systemmonitor als eigener Thread realisiert.

Für den `SystemMonitor` kann mit der Methode `setTimeout` ein Intervall definiert werden, nach dem die Methode `doMonitoring`, die von einem konkreten Systemmonitor implementiert werden muss, aufgerufen wird, um die gewünschten Parameter abzufragen und in den `SystemMonitorInformationPool` zu hinterlegen.

3.3.8 Asynchrones Laden von Klassen und Ressourcen

Verlegbare Dienste der Middleware können unter bestimmten Umständen auf einen anderen Knoten verlegt werden. Soll der Dienst dort gestartet werden, muss eine Instanz seiner Klasse und aller von ihr benötigten Klassen erzeugt werden. Darüber hinaus müssen alle Ressourcen der Klasse, beispielsweise Bilder oder Definitionen für die Internationalisierung, verfügbar sein. Der Aufwand, alle möglichen Klassen von Diensten initial auf alle Knoten zu verteilen, ist für die Middleware weder gerechtfertigt noch gewünscht. Außerdem ist das dynamische Nachladen von Klassen durch einen `ClassLoader` [6] in Java konzeptionell verankert.

Ein eigener Klassenlader als Ableitung der Klasse `ClassLoader` kann das Vorgehen beim Laden von Klassen verändern und erweitern, um beispielsweise von einem entfernten Rechner Klassen zur Laufzeit nachzuladen. Ähnliche Implementierungen existieren bereits und ermöglichen es, Klassen von einem Rechner zu laden, der per TCP/IP erreicht werden kann. Es wird dazu eine Socket-Verbindung geöffnet, über die dann, wie bisher auch, synchron die gewünschten Klassen geladen werden können.

Da die Middleware jedoch ausschließlich über eine asynchrone Kommunikation mittels Nachrichten verfügt, können die bisherigen Implementierungen nicht angewandt werden. Für den Fall einer asynchronen Kommunikation gibt es bisher noch keine Implementierung, da der Klassenlader die geforderten Klassen immer in einem Frage-Antwort-Verhalten ohne Unterbrechung lädt. Das bedeutet, dass der Klassenlader erst dann wieder verlassen wird, wenn er entweder alle notwendigen Klassen laden konnte, oder mit einer `ClassNotFoundException` abbricht.

Bei der asynchronen Kommunikation der Knoten, muss dieses Verhalten aufgebrochen werden. Wenn der Klassenlader eine Klasse anfordert, wird eine Nachricht an den Ursprungsknoten geschickt und auf die Antwort gewartet, bevor mit der Erzeugung der Klasse fortgefahren werden kann. Dieses Verhalten wurde in der Middleware implementiert. Abbildung 3.15 zeigt die Klassen, die dafür benötigt werden. Nachfolgend wird der Ablauf des asynchronen Ladens einer Klasse erläutert.

Ablauf des asynchronen Klassenladers

Wird ein Dienst von einem Knoten auf einen anderen Knoten verlegt, sendet der `Configurator` des Ursprungsknotens eine Nachricht an den Zielknoten mit dem Namen der Klasse und dem serialisierten Inhalt des Dienstes. Der `ConfiguratorService` des Zielknotens nimmt diese Nachricht entgegen und leitet sie an die Methode `createRelocatableService` des `Configurator` weiter. Der `Configurator` des Zielknotens entnimmt der

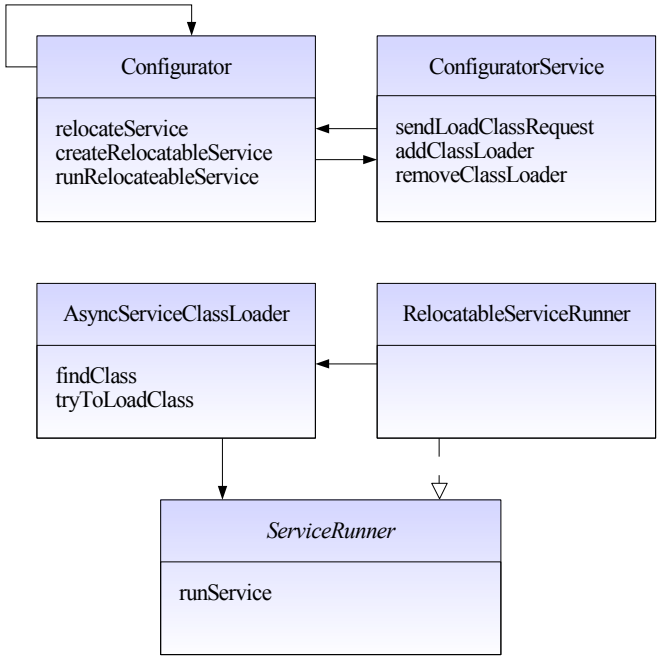

Abbildung 3.15: Klassendiagramm des asynchronen Klassenladers

Nachricht den Namen der Klasse mittels der eine Instanz des Dienstes erstellt werden kann und übergibt diesen an einen `RelocatableServiceRunner`. Dieser erstellt zuerst einen `AsyncServiceClassLoader` und übergibt dem `ConfiguratorService` eine Referenz darauf, damit dieser eintreffende Nachrichten an den passenden `AsyncServiceClassLoader` weitergeben kann. Anschließend ruft der `RelocatableServiceRunner` die Methode `tryToLoadClass` des `AsyncServiceClassLoader` auf, um das Nachladen der Dienstklasse zu initiieren. Der `RelocatableServiceRunner` wird in einem eigenen Thread ausgeführt, der auch vom `AsyncServiceClassLoader` benutzt wird, um aus den nachgeladenen Klassen eine Instanz des Dienstes zu erstellen.

Der `AsyncServiceClassLoader` versucht mit der Methode `newInstance` eine Instanz des Dienstes zu erstellen. Dadurch wird das Laden der notwendigen Klassen auf dem lokalen Knoten gestartet. Wenn alle notwendigen Klassen lokal verfügbar sind, kann die Instanz erstellt werden und der Klassenlader kann beendet werden.

Sind Klassen oder Ressourcen nicht lokal verfügbar und somit vom primären Klassenlader nicht erreichbar, ruft dieser beim `AsyncServiceClassLoader` die Methode `findClass` auf. Dort wird der `ConfiguratorService` mit der Methode `sendLoadClassRequest` angewiesen vom Ursprungsknoten die gesuchte Klasse per Nachricht anzufordern. Anschließend wird der Thread des `AsyncServiceClassLoader` angehalten und gewartet, bis der `ConfiguratorService` eine Antwort vom Ursprungsknoten erhalten hat.

Der `AsyncServiceClassLoader` wartet maximal fünf Sekunden und fordert dann die Klasse erneut vom `ConfiguratorService` an. Das Ganze wird maximal drei mal wiederholt, was einer Gesamtverzögerung von 20 Sekunden entspricht, bevor der `AsyncServiceClassLoader` das Laden der Klasse mit einer `ClassNotFoundException` abbricht.

Empfängt der `ConfiguratorService` innerhalb dieser Zeit die gewünschte Klasse, fügt er die Daten der Klasse mit der Methode `addClassByteArray` dem `AsyncServiceClassLoader` hinzu und weckt diesen durch Aufruf von `notify` wieder auf. Der `AsyncServiceClassLoader` erzeugt aus dem Byte-Array eine Klassendefinition und gibt diese zurück. Dieser Ablauf wiederholt sich so lange, bis alle notwendigen Klassen vom Ursprungsknoten geladen wurden. Dann gibt der `AsyncServiceClassLoader` eine Instanz der Klasse an den `RelocatableServiceRunner` zurück, der seinerseits den Service initialisiert, für den Start vorbereitet und letztendlich startet.

3.4 Fazit

Die Organic Ubiquitous Middleware nutzt mit den typisierten Nachrichten einen sehr flexiblen Ansatz für den Informationsaustausch. Einerseits wird das Ersetzen von Diensten zur Laufzeit ermöglicht, ohne abhängige Dienste neu kompilieren zu müssen, andererseits wird damit die Integration der Selbst-X-Eigenschaften realisiert, die ein intensives Monitoring voraussetzt.

Die klare Trennung der Middleware von der Kommunikationsinfrastruktur und die Fähigkeit, durch Erweiterung um zusätzliche TransportConnectoren, neue Kommunikationsmechanismen einfach zu integrieren, ermöglichen den Einsatz der Middleware in vielen Umgebungen. Hinzu kommt, dass die Anforderungen an eine Kommunikationsinfrastruktur, durch den asynchronen Nachrichtenaustausch, minimal sind.

Im Vergleich zu herkömmlichen Middlewaresystemen beruht die Organic Ubiquitous Middleware auf dem Observer-Controller-Architekturkonzept. Der Organic Manager stellt die Komponenten für die Verwaltung und Verteilung der Informationen aus den Monitoren bereit.

Applikationen auf Basis der Organic Ubiquitous Middleware müssen in Form von Diensten implementiert werden, die entweder auf einem Knoten laufen oder als RelocatableService zur Laufzeit auf verschiedene Knoten verlegt werden können. Die verschiebbaren Dienste stellen die Grundlage für die Umsetzung der Selbstkonfiguration und Selbstoptimierung dar.

4 Selbstkonfiguration

Die Komplexität moderner Computersysteme ist inzwischen so hoch, dass die Systeme teilweise von mehreren Personen verwaltet werden müssen, beziehungsweise die Unterstützung von zusätzlichen Computern für die Überwachung und Administration notwendig ist. Da die Rechenleistung in etwa dem gleichen Maß wie bisher wachsen wird, verschärft sich die Problematik der steigenden Komplexität weiter.

Dieses Problem trifft jedoch nicht nur auf große monolithische Systeme zu, sondern auch auf verteilte Systeme und insbesondere auf ubiquitäre Systeme, wie sie durch die bereits beschriebene Middleware unterstützt werden soll. Ubiquitäre Systeme [86] zeichnen sich durch eine große Anzahl unterschiedlicher Geräte aus, die sich in ihren Ressourcen zum Teil stark unterscheiden können. Auch wenn die Middleware zum aktuellen Stand noch keine Ad-hoc-Fähigkeit bietet und nicht für Kleinstgeräte, wie beispielsweise Mobiltelefone oder PDAs vorgesehen ist, so wird doch dem Umstand Rechnung getragen, dass auch in vernetzten Umgebungen eine Vielzahl an Computern vorhanden sein kann, die nicht einzeln administriert werden können.

Ein Ansatz zur Lösung dieses Komplexitätsproblems, wie sie auch durch die Initiativen des Autonomic Computing [30] und Organic Computing [79] verfolgt werden, ist die Nutzung verschiedener Mechanismen zur Selbstorganisation [20]. In der Natur ist Selbstorganisation ein integraler Bestandteil vieler Organismen und Populationen. Die Selbstorganisation kann durch Kombination einzelner Mechanismen wie beispielsweise Selbstkonfiguration und Selbstoptimierung erreicht werden, wobei die Wirkweisen der einzelnen Mechanismen unterschiedliche Teilziele verfolgen können.

Die Selbstkonfiguration, als ein Teilaspekt der Selbstorganisation, beschreibt die Fähigkeit eines Systems, sich ohne menschliches Zutun so zu konfigurieren, dass Vorgaben und Anforderungen für eine Konfiguration eingehalten werden. Dies kann zum einen bei der initialen Konfiguration eines Systems, zum anderen beim Entfernen oder Hinzufügen von Komponenten geschehen.

Der Begriff der Selbstkonfiguration wird häufig mit dem Begriff der Adaptivität oder Selbstadaptivität gleichgesetzt, umfasst jedoch nicht denselben funktionalen Umfang. Die Selbstkonfiguration wird genutzt, um Komponenten eines Systems und deren Wechselwirkung so zu beeinflussen, dass ein gegebenes Ziel, die Konfigurationsaufgabe, unter gewissen Randbedingungen bestmöglich erfüllt wird. Die

Selbstadaptivität hingegen geht einen Schritt weiter und beschreibt zusätzlich die Fähigkeit eines Systems sich funktional so weit zu verändern, dass es auch bei veränderten Randbedingungen die gestellte Aufgabe erfüllen kann.

In dem folgenden Kapitel wird ein dezentraler Selbstkonfigurationsmechanismus vorgestellt, der es erlaubt, eine initiale Konfiguration auf einem verteilten System so zu erstellen, dass die gegebenen Randbedingungen der Konfigurationsbeschreibung eingehalten werden und zusätzlich eine möglichst gute Verteilung der Dienste auf den vorhandenen Knoten erreicht wird. Im darauf folgenden Kapitel wird dann ein Ansatz zur Selbstoptimierung vorgestellt, der auf Basis der initialen Konfiguration das System zur Laufzeit anhand gegebener Parameter weiter optimiert.

In diesem Kapitel soll lediglich die initiale Konfiguration betrachtet werden und nicht die Möglichkeit, mit dem gleichen Verfahren auch eine Rekonfiguration des Systems beim Hinzufügen oder Entfernen von Komponenten zu erzielen.

Das vorliegende Kapitel ist wie folgt aufgebaut. Zunächst wird anhand eines Beispiels die Idee der Selbstkonfiguration durch ein kooperatives soziales Verhalten [74] vorgestellt. Daraus werden die einzelnen Schritte der Selbstkonfiguration abgeleitet. Anschließend werden die notwendigen Grundlagen und der Selbstkonfigurationsmechanismus erklärt. Danach werden die Ergebnisse der Simulation vorgestellt und die Integration in die Middleware beschrieben.

Teile der Selbstkonfiguration wurden in der Diplomarbeit „Selbstkonfiguration in einem dienstbasierten Peer-to-Peer Netzwerk“ [31] erarbeitet. In der Diplomarbeit wurden Teile der Konfigurationsbeschreibungssprache sowie der Simulator zur Evaluation der Selbstkonfiguration implementiert.

4.1 Selbstkonfiguration durch kooperatives soziales Verhalten

Der vorgestellte Algorithmus zur Realisierung der Selbstkonfiguration ist am Beispiel des kooperativen Verhaltens sozialer Gruppen mit gemeinsamen Zielen angelehnt. Ein Beispiel soll verdeutlichen, wie die einzelnen Elemente (Personen) interagieren, um ohne zentrale Kontrollinstanz zu einer optimalen Lösung des Problems zu kommen. Aus dem Beispiel werden anschließend die Schritte des kooperativen Algorithmus extrahiert, die zur Umsetzung der Selbstkonfiguration notwendig sind.

4.1.1 Beispiel für ein kooperatives soziales Verhalten

Ein Softwareunternehmen hat einen Auftrag zur Erstellung einer umfangreichen Softwarelösung erhalten. Die Anforderungen wurden in Abstimmung mit

dem Kunden geklärt und die Randbedingungen zur Erstellung der Software (Hardeware-Plattform, Programmiersprache, Benutzerschnittstelle, etc.) wurden festgelegt.

Aus diesen Anforderungen wurde ein Modulkonzept entwickelt, das die Software in einzelne Arbeitspakete aufspaltet. Die Arbeitspakete sind beispielsweise unterschieden nach Umfang, notwendigem Aufwand zur Implementierung, Komplexität und Art der notwendigen Vorkenntnisse. All diese Arbeitspakete werden mit den entsprechenden Anforderungen auf eine Liste geschrieben.

In der Projektbesprechung bringt der Projektleiter diese Liste mit und lädt alle Entwickler zur Besprechung ein. Ziel der Projektbesprechung ist die Verteilung der Arbeitspakete auf die Entwickler in Abhängigkeit ihrer Fähigkeiten, so dass das Projekt möglichst kompetent abgewickelt werden kann. Dabei ist zu beachten, dass nicht jeder Entwickler über die gleichen Fähigkeiten und Fertigkeiten verfügt. Während sich die Designer fast ausschließlich mit dem Design der Benutzeroberfläche befassen, sollten sich die Programmierer stärker um die Implementierung der Funktionalität kümmern.

Da der Projektleiter nicht ständig den aktuellen Wissensstand seiner Mitarbeiter erfragen und objektiv bewerten kann, überlässt er es den Mitarbeitern sich für die einzelnen Arbeitspakete zu entscheiden. Dabei ist von besonderer Bedeutung, dass alle die gleiche firmeninterne Bewertungsgrundlage ihrer Qualifikationen heranziehen.

Nachdem sich alle im Besprechungsraum eingefunden haben, verteilt der Projektleiter an jeden Mitarbeiter die Liste mit den Arbeitspaketen. Jeder Mitarbeiter streicht zunächst alle Arbeitspakete, für die er nicht qualifiziert ist, bewertet anschließend die verbleibenden Arbeitspakete auf Grundlage seiner Qualifikationen und sortiert diese in absteigender Reihenfolge. Anstatt dass nun der Projektleiter jedes Arbeitspaket einzeln für eine Belegung aufruft, beginnt einer der Mitarbeiter ein Arbeitspaket auszuwählen, das er am besten erledigen kann und nennt das Arbeitspaket sowie die berechnete Qualität mit der er dieses erbringen kann. Alle anderen hören zu und markieren auf ihrer Liste das entsprechende Arbeitspaket als zugewiesen. Grundsätzlich gilt bei der Firma, dass man nicht durcheinander redet, sondern den Anderen aussprechen lässt und dann mögliche Verbesserungsvorschläge in eine Diskussion einbringt.

Es kann vorkommen, dass für die Fertigstellung eines Arbeitspaketes mehrere Mitarbeiter benötigt werden. In diesem Fall werden bei dem entsprechenden Arbeitspaket alle Mitarbeiter eingetragen, die daran arbeiten werden.

Hat ein Mitarbeiter festgestellt, dass er das gleiche Arbeitspaket besser erbringen könnte, nennt er einfach das gleiche Arbeitspaket, aber mit seiner besseren Qualität. Die anderen überschreiben daraufhin die bereits bestehende Zuweisung und tragen anstelle des ersten Mitarbeiters den zweiten Mitarbeiter mit seiner höheren Qualität ein. Der Mitarbeiter, der überschrieben wurde, trägt ebenfalls

den anderen Mitarbeiter ein und kann sich jetzt für eine andere Aufgabe entscheiden.

Da keiner der Mitarbeiter weiß, wann sich der nächste Mitarbeiter für eine der Aufgaben entscheidet, kann auch der Zeitpunkt, zu dem die nächste Belegung einer Aufgabe erfolgt, nicht vorherbestimmt werden. Darum kann es passieren, dass zwei Mitarbeiter fast gleichzeitig eine Belegung bekannt geben. Da alle anderen Mitarbeiter aufmerksam zuhören, können sie die beiden Belegungen der Arbeitspakete mithören.

Haben sich die beiden Mitarbeiter für unterschiedliche Arbeitspakete entschieden, können alle die Belegungen einfach eintragen. Ist die Entscheidung bei beiden auf das gleiche Arbeitspaket gefallen, entscheidet die Angabe der Qualität, wer in die Liste übernommen wird und wer nicht. Selbst die beiden Mitarbeiter, die gleichzeitig ihre Belegungen bekannt gegeben haben, können anhand der Qualität entscheiden, wer letztendlich in die Liste für das Arbeitspaket eingetragen wird.

Ein Problem für die Belegung der Arbeitspakete tritt nur dann auf, wenn die beiden Mitarbeiter das Arbeitspaket mit der gleichen Qualität erbringen können. Da sie gleichzeitig die Belegung bekannt gegeben haben, kann zunächst nicht entschieden werden, wer nun das Arbeitspaket zugewiesen bekommt. In diesem Fall ist eine Konfliktlösung notwendig, die anhand zusätzlicher Kriterien, wie beispielsweise die Anzahl der bereits belegten, oder noch zur Belegung möglichen Arbeitspakete des Mitarbeiters, eindeutig entscheidet, wer das Arbeitspaket erhält. Der Vorteil dieser Strategie ist zum einen, dass keine weitere Diskussion zwischen den beiden Mitarbeitern notwendig ist, und dass jeder Mitarbeiter die gleiche Entscheidung auch zu einem späteren Zeitpunkt treffen kann.

Wurden alle Arbeitspakete zugewiesen, vergewissert sich der Projektleiter noch einmal, ob auch alle die gleiche Zuordnung der Arbeitspakete vorgenommen haben, um zu vermeiden, dass manche Arbeitspakete gar nicht, oder fälschlicher Weise mehrfach erledigt werden. Dazu übergibt er jedem Mitarbeiter eine Kopie seiner Liste der Arbeitspakete mit den eingetragenen Arbeitspaketen. Jeder Mitarbeiter kann nun die Eintragungen abgleichen und für den Fall, dass es unterschiedliche Eintragungen gibt, aus beiden Listen die besten Zuordnungen ermitteln. Unterschiedliche Belegungen können auftreten, wenn ein Mitarbeiter eine Belegung überhört hat und somit keinen, oder einen anderen Eintrag in seiner Liste hat. Ist ein Eintrag in der Liste des Mitarbeiters besser als auf der Liste des Projektleiters, so hat wohl der Projektleiter irgendwann eine Belegung überhört und der Mitarbeiter teilt den anderen die bessere Belegung mit, indem er seine Liste an alle anderen zum Abgleichen verteilt.

Mit dieser letzten Runde wird sichergestellt, dass alle Mitarbeiter über die gleichen Informationen verfügen und somit die Arbeit am Projekt aufgenommen werden kann.

4.1.2 Schritte der kooperativen Lösungsfindung

Aus der Beschreibung des Beispiels lassen sich verschiedene Phasen bei der Zuordnung der Arbeitspakete identifizieren. Darüber hinaus sind auch Probleme erkennbar, die durch den dezentralen Ansatz des Algorithmus entstehen. Da keine zentrale Instanz den Ablauf steuert, der Projektleiter bringt lediglich die Listen mit und initiiert die abschließende Verifikationsphase, kann es zu zeitlichen Abhängigkeiten bei der Belegung der Arbeitspakete kommen, die bei der Umsetzung im Selbstkonfigurationsalgorithmus beachtet und aufgelöst werden müssen. Der Ablauf kann in drei Phasen unterteilt werden.

1. *Verteilungsphase*
   Die Liste der Arbeitspakete wird an alle Mitarbeiter verteilt und jeder Mitarbeiter streicht auf seiner Liste alle Arbeitspakete, die er mangels Qualifikation nicht erbringen kann. Anschließend bewertet er die verbliebenen Arbeitspakete und sortiert diese in absteigender Reihenfolge nach der Qualität, mit der diese erbracht werden können.

2. *Zuweisungsphase*
   Die Mitarbeiter tragen sich jeweils bei dem Arbeitspaket ein, das sie mit der besten Qualität erledigen können und teilen den anderen mit, um welches Arbeitspaket es sich handelt und mit welcher Qualität dieses Arbeitspaket erbracht werden kann.

   Entscheidet sich ein Mitarbeiter dazu, die Belegung eines anderen Mitarbeiters zu überschreiben, weil er das Arbeitspaket mit einer höheren Qualität erbringen kann, teilt er dies den anderen Mitarbeitern nach dem gleichen Verfahren wie bei einer Neubelegung mit. Werden für ein Arbeitspaket mehrere Mitarbeiter benötigt, werden so lange neue Belegungen bei dem Arbeitspaket eingetragen, bis die notwendige Anzahl erreicht wurde. Kommt es danach zu einer weiteren Belegung für dieses Arbeitspaket, wird der Mitarbeiter mit der geringsten Qualität von der Liste gestrichen.

   In dieser Phase kann es zu Konflikten kommen, wenn beispielsweise mehrere Mitarbeiter gleichzeitig eine Belegung bekannt geben oder weil zwei Mitarbeiter ein Arbeitspaket mit der gleichen Qualität erbringen können. Der Algorithmus muss in der Lage sein, diese Konflikte ohne zusätzliche Kommunikation zwischen den Konfliktpartnern aufzulösen.

3. *Verifikationsphase*
   Um sicher zu stellen, dass bei allen Mitarbeitern die gleichen Belegungen existieren, gibt der Projektleiter allen Mitarbeitern seine Belegungsliste, die jeder Mitarbeiter mit seiner Liste abgleicht. Erkennt der Mitarbeiter eine bessere Belegung auf der Liste des Projektleiters, passt er seine Belegung

an. Hat der Mitarbeiter eine bessere Belegung für mindestens ein Arbeitspaket in seiner Liste gefunden, verteilt er seine Liste, damit alle anderen Mitarbeiter gegebenenfalls die verbesserte Belegung übernehmen können. Zu solchen Inkonsistenzen bei der Belegung kann es kommen, wenn beispielsweise eine Belegung verpasst wurde.

Aus den genannten Phasen wird nachfolgend der Algorithmus zur Selbstkonfiguration abgeleitet und auf die Bedürfnisse der Middleware angepasst. Der Selbstkonfigurationsalgorithmus soll die Konflikte, die während der Zuweisungsphase entstehen können, ohne zusätzliche Nachrichten auflösen. Hierzu bedarf es einer Konfliktlösungsstrategie, die anhand zusätzlicher Informationen eine eindeutige Zuweisung berechnen kann.

Da es sich bei dem Ansatz um einen verteilten Algorithmus handelt, muss gewährleistet werden, dass Verzögerungen bei der Nachrichtenauslieferung, wie sie durch Latenzzeiten von Netzwerken oder stark belasteten Knoten entstehen können, keinen negativen Einfluss auf die Konsistenz und die Qualität des Ergebnisses haben. Somit ist auch sichergestellt, dass die Reihenfolge, in der Nachrichten bei den Knoten eintreffen, die Entscheidung für die Belegung nicht beeinflussen.

Das Problem der Zuweisung von Aufgaben zu einzelnen Personen unter Berücksichtigung von Randbedingungen ist eine Aufgabe der Problemklasse NP-hart [51], was bedeutet, dass es keinen deterministischen Algorithmus zur Berechnung der optimalen Lösung gibt, und dass alle Algorithmen, die das Problem lösen mindestens eine polynominale Laufzeit aufweisen.

In der Klasse der Distributed Constraint Satisfaction Problems (DSCP)[91, 40] werden verteilte Algorithmen betrachtet, die Lösungen für ähnliche Probleme wie das der Selbstkonfiguration lösen.

Der Ansatz der vorgestellten Selbstkonfiguration versucht die Komplexität der DSCP-Algorithmen zu vermeiden und bewusst einen einfachen Ansatz zu verfolgen, da es für die gestellte Aufgabe der Erstkonfiguration nicht erforderlich ist eine optimale Lösung zu erreichen. Die Erstkonfiguration soll eine möglichst gute Verteilung der Aufgaben unter Berücksichtigung der vorhandenen Ressourcen ermöglichen und als Ausgangsbasis für die nachgelagerte Selbstoptimierung dienen. Da die Selbstoptimierung in der Lage ist, Dienste von einem Knoten des Netzwerks auf andere Knoten zu verlagern, wäre der Aufwand für die Berechnung der optimalen Lösung für die Erstkonfiguration nicht gerechtfertigt, zumal meist keine exakten Angaben über die benötigten Ressourcen gemacht werden können. Die Selbstoptimierung ermittelt den tatsächlichen Bedarf der einzelnen Dienste zur Laufzeit und erzeugt somit ein besseres Anpassungsverhalten an die realen Gegebenheiten.

4.2 Modellbildung

Die Umsetzung der Selbstkonfiguration erfordert die Definition verschiedener Elemente der Middleware, die zum einen der Beschreibung der Konfiguration dienen, zum anderen die Berechnung der Dienstgüte ermöglichen.

Die Liste der Arbeitspakete wird durch eine Konfigurationsbeschreibung ersetzt, in der die Dienste und Monitore der Applikation enthalten sind. Die Dienste sollen anhand des angegebenen Ressourcenbedarfs auf die Knoten der Middleware verteilt werden. In der Konfiguration können darüber hinaus Bedingungen (Constraints) angegeben werden, die alle Knoten prüfen müssen, um bei positiver Bewertung den angegebenen Dienst oder Monitor einzubinden. Für die Berechnung der Dienstgüte wird eine Metrik definiert, die für alle Knoten der Middleware eine reproduzierbare Entscheidungsgrundlage bei der Beurteilung von Zuweisungen bietet.

4.2.1 Konfigurationsbeschreibung

Die Beschreibung der Konfiguration erfolgt in der Auszeichnungssprache XML [84]. Die leichte Erweiterbarkeit der Sprache ermöglicht eine einfache Anpassung an veränderte Anforderungen. XML-Dokumente sind in menschenlesbarer Form, außerdem existieren für nahezu alle Plattformen XML-Parser, mit denen XML-Dokumente in eine maschineninterne Darstellung überführt werden können. Im Fall der Middleware werden die XML-Dokumente in die Objektstruktur einer Konfiguration übersetzt. Die XML-Schemadefinition ist in Anhang A abgebildet. Mit dem XML-Schema kann die syntaktische Korrektheit einer Konfigurationsdatei überprüft werden.

Der folgende Absatz zeigt die Struktur einer Konfigurationsdatei mit den grundlegenden Elementen für die Bedingungen (`constraints`), Dienste (`service`) und Monitore (`monitor`). Bedingungen werden benutzt, um bestimmte Dienste oder Monitore auf alle Knoten zuzuweisen, die über die entsprechenden Voraussetzungen an Ressourcen verfügen. Dienste und Monitore werden mit den entsprechenden Tags beschrieben, wobei jeweils ihre benötigten Ressourcen angegeben werden können. Die drei Elemente `constraints`, `service` und `monitor`, sowie die Definition von Ressourcen werden in den folgenden Abschnitten erklärt.

```
<?xml version="1.0" encoding="UTF-8"?>
  <configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:noNamespaceSchemaLocation="configuration.xsd">
    <minimumConfiguration>
      <constraints>
        <forall>
          <having>
            <resource><value></value></resource>
          </having>
          <provide>
            <monitor><ressource><value></value></ressource></monitor>
            <service><ressource><value></value></ressource></service>
          </provide>
        </forall>
        </constraints>
        <service><ressource><value></value></ressource></service>
        <monitor><ressource><value></value></ressource></monitor>
      </minimumConfiguration>
  </configuration>
```

Ressourcen

Der Tag `resource` wird zur Beschreibung von Ressourcen benötigt. Ressourcen können bei den Anforderungen von Diensten und Monitoren, aber auch in einer Bedingung benutzt werden. Das folgende Beispiel definiert eine Ressource `CPU` und einen zugehörigen Wert mit der Bezeichnung `frequency` und der Einheit `Mhz`. Eine Ressource kann auch durch mehrere Werte beschrieben werden, wobei dann für die Ressource mehrere `value` Tags angegeben werden.

```
<resource name="CPU">
  <value name="frequency" unit="Mhz">1450</value>
</resource>
```

Ressourcen sind frei definierbar und werden bei der Berechnung der Dienstgüte benötigt. Die definierten Ressourcen müssen jedoch bekannt sein, da sonst die Berechnung der Dienstgüte unmöglich wird.

Die freie Definition der Ressourcen besitzt den Nachteil, dass für eine Ressource unterschiedliche Bezeichnungen benutzt werden können. Es kann sogar vorkommen, dass für ein und dieselbe Ressource, aufgrund einer Mehrdeutigkeit der Definition, die Werte nicht direkt vergleichbar sind. Das obige Beispiel stellt einen solchen Fall dar.

Die Angabe der Taktfrequenz einer CPU alleine ist nicht ausreichend, um die Leistungsfähigkeit eines Prozessors zu definieren, da diese stark von der Prozessorfamilie beziehungsweise der verwendeten Prozessorarchitektur abhängt. Solche

Mehrdeutigkeiten müssen auf einer übergeordneten semantischen Ebene gelöst werden.

Ein Ansatz zur Lösung eines der beiden Probleme ist die Nutzung von Normalisierungsfaktoren, die im obigen Beispiel alle Arten von Prozessoren auf einen Einheitsprozessor abbilden könnten. Auf dieser Ebene wären dann die Werte wieder vergleichbar.

Ein Ansatz zur Lösung semantischer Mehrdeutigkeiten bietet die Web Ontology Language (OWL) [83]. Ontologien werden benutzt, um semantische Gleichheit bei unterschiedlichen Bezeichnungen zu erkennen. In Abhängigkeit eines gegebenen Kontextes für den eine Ontologie definiert ist, können verschiedene Bezeichnungen der gleichen Bedeutung zugeordnet und somit vergleichbar gemacht werden.

Beide Ansätze werden nicht weiter untersucht, da die Definition der Konfigurationsbeschreibung möglichst einfach gehalten werden soll und zudem keine optimale Lösung gefunden werden muss. Die initiale Konfiguration wird durch die Selbstoptimierung zur Laufzeit wieder verändert und anhand real gemessener Werte weiter optimiert.

Dienste und Monitore

Applikationen auf Basis der Middleware bestehen aus einer Anzahl von Diensten und Monitoren, die miteinander interagieren, um dem Benutzer den Eindruck einer Gesamtapplikation zu vermitteln. Für den Benutzer ist die Verteilung der Dienste und Monitore im Netzwerk vollkommen transparent. Die Verteilung kann sich gegebenenfalls zur Laufzeit verändern.

Dienste und Monitore stellen somit die Elemente der Konfiguration dar, die durch die Selbstkonfiguration im Netzwerk, anhand der gegebenen Ressourcenanforderungen, verteilt werden. Der folgende Auszug aus einer Konfigurationsanforderung zeigt Beispiele für einen Dienst und einen Monitor.

```
<service id="2" amount="2" name="DataBase"
         class="de.uau.SqliteBinding">
  <resource name="RAM">
    <value name="size" unit="MB">256</value>
  </resource>
  <resource name="CPU">
    <value name="frequency" unit="Mhz">1450</value>
  </resource>
</service>
```

```
<monitor id="1" amount="1" name="Surveillance System"
         class="de.uau.SrvMonitor">
  <resource name="IR SENSOR">
    <value name="range" unit="m">5</value>
  </resource>
</monitor>
```

Dienste und Monitore sind in der Konfigurationsbeschreibung nach dem gleichen Schema aufgebaut und besitzen die gleichen Attribute. Aus diesem Grund werden Dienste und Monitore, in Anlehnung an das Beispiel der Firma, auch als Jobs bezeichnet. Die Attribute eines Jobs sind:

- `id`
  Eine Identifikation, um den Job innerhalb einer Konfigurationsspezifikation eindeutig zu identifizieren. Die Identifikation wird bei der Belegung von Jobs benutzt, um den anderen Knoten mitzuteilen, um welchen Job es sich handelt.

- `amount`
  Die Anzahl der Instanzen, die von dem Job im Netzwerk erstellt werden soll. Werden von einem Job mehrere Instanzen benötigt, wird für den Job eine Providerliste erstellt, in der alle Knoten (Provider) eingetragen werden, die eine Instanz des Jobs belegen.

- `name`
  Eine vom Menschen lesbare Bezeichnung des Jobs. Der Name ist für die Selbstkonfiguration nicht von Bedeutung, sondern dient lediglich der Übersichtlichkeit.

- `class`
  Die Klasse, die den Job implementiert. Da die Middleware in Java implementiert ist, kann mit dieser Referenz von jedem beliebigen Knoten aus eine Instanz der Klasse durch den `AsyncClassLoader` erstellt werden (siehe 3.3.8).

Ein Job hat neben seinen Attributen mindestens eine Ressource, möglicherweise auch mehrere Ressourcen, die zur Berechnung der Dienstgüte des Jobs von den Knoten benutzt werden.

Bedingungen

Eine Besonderheit ubiquitärer Systeme ist die große Anzahl unterschiedlicher Elemente, aus denen sich das System zusammensetzt. Mit den bereits beschriebenen Elementen Ressource und Job beziehungsweise Service und Monitor, kann

eine Applikation mit ihren grundlegenden Bestandteilen beschrieben werden, um sie auf den vorhandenen Knoten verteilen zu können. Für den Fall, dass jedoch mehrere gleichartige Komponenten im System vorhanden sind, die alle in die Applikation mit einbezogen werden sollen, beispielsweise Sensoren, ist die bisherige Beschreibung nicht mächtig genug.

Die Erweiterung der Konfiguration um Bedingungen (Constraints) ermöglicht die Definition von Abhängigkeiten zwischen vorhandenen Ressourcen und den zu erbringenden Jobs. Beispielsweise kann damit auf jedem Knoten, der über einen Sensor eines Location-Tracking-Systems verfügt, der zugehörige Dienst oder Monitor gestartet werden, um die angeschlossene Hardware zu nutzen. Ein weiteres Beispiel entstammt dem Smart-Doorplate-Projekt, bei dem an verschiedenen Knoten ein Touchscreen angeschlossen ist. Jeder Knoten, der über einen Touchscreen verfügt, soll den Smart-Doorplate-Dienst starten.

Die Beschreibung dieser Zusammenhänge in natürlicher Sprache erinnert an den mathematischen Allquantor: „Für alle Knoten, die eine bestimmte Ressource besitzen, soll ein gegebener Dienst gestartet werden“. Ein ähnliches Konstrukt ist auch in der Object Constraint Language [43] definiert. Der folgende Auszug aus einer Konfigurationsspezifikation zeigt ein Beispiel für eine Bedingung, bei der ein Monitor in Abhängigkeit von einem Infrarotsensor gestartet werden soll.

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

Eine Bedingung wird durch den Tag `constraints` gekapselt. Darin enthalten ist der Tag `forall`, der einen Tag `having` besitzt, in dem die Voraussetzungen zur Erfüllung der Bedingung beschrieben werden und einen Tag `provide`, der angibt, welche Jobs gegebenenfalls von dem Knoten erbracht werden sollen. Innerhalb des `having` Tags, der die Voraussetzungen beschreibt, können Ressourcen spezifiziert werden. Es muss mindestens eine Ressource angegeben werden.

Bedingungen werden im Verlauf der Selbstkonfiguration immer als erstes geprüft, bevor die Verhandlung über die Belegung der restlichen Jobs beginnt. Werden

aufgrund von Bedingungen bereits Jobs zu einem Knoten hinzugefügt, reduzieren sich die Ressourcen des Knotens um den entsprechenden Ressourcenbedarf der Jobs.

Eine Erweiterung der Bedingungen, die zu Beginn der Selbstkonfiguration geprüft werden, kann benutzt werden, um Abhängigkeiten zwischen Diensten zu definieren, die während der Belegung der Jobs als zusätzliche Parameter zur Berechnung der Dienstgüte herangezogen werden. Eine Bedingung oder Abhängigkeit (`dependency`) dieser Form könnte beispielsweise die Belegung zweier Dienste so beeinflussen, dass sie gemeinsam auf einem Knoten belegt werden oder auf zwei Knoten, die durch eine entsprechend gute Kommunikationsverbindung miteinander gekoppelt sind.

```
<constraints>
  <dependency>
    <jobs>
      <id>1</id>
      <id>3</id>
    </jobs>
    <alternative>
      <onSameNode>
    </alternative>
    <alternative>
      <resource name="Network" >
        <value name="Bandwidth" unit="GBit">1</value>
      </resource>
    </alternative>
  <dependency>
</constraints>
```

Abhängigkeiten werden so definiert, dass zunächst die betroffenen Jobs und anschließend die Alternativen zur Befriedigung der Abhängigkeiten definiert werden. Bei der Berechnung der Dienstgüte wird die Dienstgüte jeder Alternative geprüft und die beste Alternative für die Belegung ausgewählt.

Der Tag `<onSameNode>` erzeugt für die Dienstgüte den gleichen Wert für den abhängigen Job wie für den bereits zugewiesenen Job. Damit ist sichergestellt, dass beide Dienste auf dem gleichen Knoten belegt werden und dass bei einer Überschreibung des einen Dienstes auch eine Überschreibung des anderen Dienstes ermöglicht wird. Werden ausschließlich Ressourcen als Alternativen angegeben, wird die Berechnung der Dienstgüte lediglich durch die definierten Ressourcen als Parameter erweitert.

Mit der gegebenen Definition von Abhängigkeiten können baumartige Strukturen aufgebaut werden, die sogar Zyklen enthalten können. Da bei der Berechnung

der Dienstgüte jedoch nur die Abhängigkeiten zwischen definierten Diensten betrachtet werden, kann die Auflösung von Zyklen, wie sie bei vielen Prüfungen von Abhängigkeiten Probleme bereiten, entfallen.

4.2.2 Quality of Service Metrik

Die Dienstgüte dient den Knoten als Entscheidungsgrundlage zur Berechnung der Prioritäten der einzelnen Jobs. Je größer die Dienstgüte ist, desto besser kann ein Job von einem Knoten erbracht werden. Weiterhin wird die Dienstgüte bei der Belegung von Jobs benutzt, um den anderen Knoten mitzuteilen, wie gut ein Job erbracht werden kann. Dies gibt den anderen Knoten die Möglichkeit, eine Zuweisung zu überschreiben, für den Fall, dass ein anderer Knoten diesen Dienst besser erbringen kann.

Da der Selbstkonfigurationsalgorithmus mit sehr einfachen Methoden arbeiten soll, wird auch bei der benutzten Metrik Wert auf Einfachheit gelegt. Der einfachste Weg, die Auslastung der Ressourcen eines Knotens zu berechnen, ist, die verbrauchten Ressourcen von den noch vorhandenen Ressourcen zu subtrahieren.

In die Berechnung der Dienstgüte werden alle Ressourcen einbezogen, die in einer Jobbeschreibung enthalten sind. Damit die unterschiedlichen Wertbereiche der Ressourcen das Ergebnis der Berechnung nicht negativ beeinflussen, müssen die Werte normalisiert werden. Würden die Werte der einzelnen Ressourcen einfach addiert und durch die Gesamtzahl der Ressourcen geteilt, so würden Ressourcen mit großen Werten, wie beispielsweise die Taktfrequenz einer CPU, stärker in das Ergebnis eingehen, als beispielsweise die Reichweite eines Infrarot- oder BlueTooth-Empfängers, dessen Werte sich maximal im zweistelligen Bereich bewegen.

Sei r_i der Ressourcenbedarf eines Jobs bezüglich der Ressource i, R_i der Gesamtwert der Ressource i und rr_i der noch verfügbare Rest der Ressource i. Damit kann die Belastung b_i für die Ressource i berechnet werden.

$$b_i = \frac{rr_i - r_i}{R_i} \qquad , R_i > 0 \tag{4.1}$$

Aus der Belastung der Ressource i kann eine Dienstgüte qos_i bezüglich der Ressource i berechnet werden, indem der Wert von 1 subtrahiert wird. Der berechnete Wert gibt ein Maß für die noch freien Kapazitäten des Knotens bezüglich der Ressource i an.

$$qos_i = 1 - b_i \tag{4.2}$$

Werden mehrere Ressourcen betrachtet, so ergibt sich die Dienstgüte *qos* aus dem arithmetischen Mittel der Summe der einzelnen qos_i. Der Wert wird zusätzlich mit einem konstanten Faktor multipliziert, um den Wertbereich auf ganze Zahlen abzubilden.

$$qos = c * \frac{1}{n} \sum_{i=1}^{n} qos_i \tag{4.3}$$

Wurde ein Job einem Knoten zugewiesen, so reduziert sich der Wert rr_i um den Wert r_i. Es wäre also möglich, dass der Wert von $rr_i \leq 0$ wird, was eine Überlastung des Knotens bedeuten würde.

Für die Selbstkonfiguration wird davon ausgegangen, dass Überlasten grundsätzlich toleriert werden können. Beispielsweise steht auf einem System durch die virtuelle Speicherverwaltung mehr Speicher zur Verfügung, wie real im System vorhanden ist. Eine Überlast führt lediglich zu einer schlechteren Performance des Systems. Ähnlich verhält es sich bei der Angabe der Taktfrequenz der CPU. Für viele Programme wird eine Taktfrequenz der CPU als Anhaltspunkt für die notwendige Leistung der CPU angegeben. Wird dieser Wert unterschritten leidet zwar die Ausführungsgeschwindigkeit der Software, aber die Applikation kann grundsätzlich ausgeführt werden.

Im Fall von Überlasten soll die Dienstgüte mehr als linear abnehmen und sehr schnell große negative Werte erreichen, um Knoten, die für den Job noch eine bessere Dienstgüte angeben im Verhältnis besser zu stellen. Die Formel für den negativen Fall lautet:

$$qos_i = 1 - \frac{|rr_i| + R_i}{r_i} \qquad , r_i > 0 \tag{4.4}$$

Bei der Berechnung der Dienstgüte muss dementsprechend zwischen den beiden Fällen unterschieden werden, abhängig davon, ob rr_i positiv, oder negativ ist.

$$qos = c * \frac{1}{n} \sum_{i=1}^{n} qos_i \tag{4.5}$$

$$\text{mit} \qquad qos_i = \begin{cases} 1 - \dfrac{rr_i - r_i}{R_i} & \text{wenn } rr_i > 0, R_i > 0 \\ 1 - \dfrac{|rr_i| + R_i}{r_i} & \text{wenn } rr_i \leq 0, r_i > 0 \end{cases}$$

Formel 4.5 umfasst sowohl den Fall, dass der vorhandene Rest einer Ressource positiv ist, als auch den Fall, wenn ein Knoten bereits überlastet wurde und somit einen negativen Betrag für rr_i besitzt.

Die zusätzlichen Bedingungen, dass im ersten Fall der Maximalwert einer Ressource $R_i > 0$ sein muss, ist zwangsläufig gegeben, da die Ressource sonst nicht vorhanden wäre und somit auch nicht in der Berechnung berücksichtigt werden darf. Gleiches gilt für den zweiten Fall, bei dem gefordert wird, dass der Ressourcenbedarf r_i eines Jobs bezüglich der Ressource $i > 0$ ist, da sonst der Ressourcenbedarf keinen Einfluss auf die Dienstgüte hat, da die Ressource nicht benötigt wird.

Die Berechnung der Dienstgüte mit Formel 4.5 ist für alle Knoten des Netzwerks gleich, um einen gemeinsamen Bezugspunkt für die Bewertung der Jobs zu erhalten. Die Formel ist zudem einfach und erfordert nur eine geringe Rechenleistung. Die Metrik ist bewusst so einfach gewählt, um zu zeigen, dass das Verfahren bereits mit sehr einfachen Mitteln sehr gute Ergebnisse produziert.

4.3 Selbstkonfigurationsprozess

Nachdem in den vorangegangenen Abschnitten die Grundlagen für den kooperativen Algorithmus zur Selbstkonfiguration erarbeitet wurden, soll nun der Algorithmus, wie er bereits aus dem Beispiel zu Beginn des Kapitels erläutert wurde, im Detail erklärt werden.

Jeder Knoten verschickt bei einer Belegung eine Nachricht. Diese Nachricht wird als Broadcast an alle anderen Knoten gesendet. Für die folgenden Betrachtungen und die spätere Evaluierung wird davon ausgegangen, dass ein Broadcast einer Nachricht entspricht, da alle Knoten durch ein Ethernet-Netzwerk verbunden sind. Würde ein anderes physikalisches Übertragungsmedium zum Einsatz kommen, wie beispielsweise BlueTooth oder WLAN, kann diese Annahme nicht aufrecht erhalten werden, da das Medium zwischen den Komponenten geteilt wird und somit jeder Knoten durch eine eigene Nachricht informiert werden müsste.

4.3.1 Verteilung der Konfigurationsanforderung

Zu Beginn der Selbstkonfiguration wird die Applikation, bestehend aus den Klassendateien und der Konfigurationsanforderung, auf einem Knoten durch den Administrator hinterlegt. Nach erfolgreicher Konfiguration können alle anderen Knoten die notwendigen Klassen von diesem Knoten laden.

In der ersten Phase wird die Konfigurationsspezifikation an alle Knoten des Netzes verteilt. Der Knoten, der die Konfigurationsanforderung erhalten hat, liest

die XML-Datei ein und erzeugt daraus ein Konfigurationsobjekt. Das Konfigurationsobjekt verteilt er dann mit einer Broadcast-Nachricht an alle Knoten.

Jeder Knoten, der das Konfigurationsobjekt erhalten hat, prüft darauf hin alle Jobs der Konfiguration und streicht die Jobs, die er aufgrund fehlender Ressourcen nicht erbringen kann.

Erhält ein Knoten die Konfiguration nicht, weil er beispielsweise erst später dem Netzwerk beitritt oder weil ein Fehler in der Kommunikation aufgetreten ist, wird er die Konfiguration anfordern, sobald er von einem anderen Knoten eine Belegungsnachricht erhält. Daraus kann er schließen, dass bereits eine Konfigurationsbeschreibung verteilt wurde und kann diese von einem beliebigen Knoten anfordern.

4.3.2 Kooperative Dienstverteilung

Der kooperative Algorithmus der Selbstkonfiguration zur Verteilung der Jobs unterteilt sich in zwei Phasen. In der ersten Phase prüft jeder Knoten die Erfüllbarkeit aller Bedingungen (Constraints) der Konfigurationsanforderung, in der zweiten Phase wird versucht, die Jobs anhand der Dienstgüte in absteigender Reihenfolge zu belegen.

Prüfen der Bedingungen

Algorithmus 1 beschreibt die erste Phase des Selbstkonfigurationsalgorithmus. Zunächst werden alle Bedingungen (Constrains) und alle Jobs der Konfigurationsanforderung ermittelt (Zeilen 1 und 2). Anschließend wird eine leere Liste für die Jobs erstellt, die potentiell erbracht werden können und eine Liste, in der die Jobs hinterlegt werden, die aufgrund fehlender Ressourcen nicht erbracht werden können (Zeilen 3 und 4).

Ab Zeile 5 wird jede Bedingung geprüft, ob sie erfüllt werden kann (Zeile 6) und alle Jobs der Bedingung werden ermittelt und zur Ausführung vorgemerkt (Zeile 7). Aufgrund der zugewiesenen Jobs muss der Ressourcenverbrauch der Jobs von den vorhandenen Ressourcen subtrahiert werden (Zeile 8).

Bewertung der Jobs

Nun wird in der Liste aller Jobs jeder Job einzeln geprüft, ob er durch den Knoten erbracht werden kann (Zeile 11 und 12). Wenn ein Job erbracht werden kann, wird mit den aktuell vorhandenen Ressourcen die Dienstgüte berechnet (Zeile 13) und der Job wird mit seiner Dienstgüte, in die anfangs leere *joblist*, eingetragen (Zeile 14). In dieser Liste werden alle Jobs eingetragen, die potentiell erbracht werden können.

Algorithmus 1 Berechnung der erfüllbaren Jobs

```
1:  constraints ← configuration.getConstraints();
2:  jobs ← configuration.getJobs();
3:  joblist ← new List();
4:  undoableJobs ← new List();
5:  for all constraint c in constraints do
6:     if canBeProvided(c) then
7:        provide(c.getServices());
8:        reduceAvailabeCapacity(c.getRequirements());
9:     end if
10: end for
11: for all job j in jobs do
12:    if canBeProvided(j) then
13:       qos ← calculateQualityOfService(j);
14:       joblist.add((qos, j));
15:    else
16:       undoableJobs.add(j);
17:    end if
18: end for
19: sort(joblist);
```

Wird bei der Prüfung in Zeile 12 festgestellt, dass der Dienst nicht erbracht werden kann, wird er in die Liste der *undoableJobs* eingetragen. In dieser Liste werden alle Jobs hinterlegt, die aufgrund fehlender Ressourcen auf dem Knoten nicht erbracht werden können.

Es werden alle Jobs in einer der beiden Listen hinterlegt, um später prüfen zu können, ob alle Jobs durch die Selbstkonfiguration auch tatsächlich verteilt werden konnten. Wird festgestellt, dass einer der Jobs nicht belegt wurde, können die Knoten über ein mehrstufiges Verfahren feststellen, ob es sich dabei um ein Kommunikationsproblem handelt, oder ob die Konfiguration tatsächlich nicht erfüllbar ist.

Abschließend wird in der ersten Phase der Selbstkonfiguration die *jobList* mit den potentiellen Jobs des Knotens in absteigender Reihenfolge bezüglich der Dienstgüte sortiert, so dass der Job mit der besten Dienstgüte als erster in der Liste steht.

Belegung der Jobs

Die Knoten wechseln während der Belegung der Jobs zwischen zwei Zuständen. Im ersten, passiven Zustand wartet der Knoten auf eingehende Nachrichten. Im

zweiten, aktiven Zustand werden die eingegangenen Nachrichten verarbeitet und versucht den nächsten Job zu belegen.

Ein Knoten startet immer im passiven Zustand und wartet auf eingehende Nachrichten. Dieser Zustand entspricht dem Zustand der Mitarbeiter, wenn sie ihren Kollegen zuhören und die Belegungen der Jobs notieren. Wenn der Knoten in den aktiven Zustand wechselt, werden die eingegangenen Nachrichten verarbeitet. Dabei werden die Dienstbelegungen in der *joblist* eingetragen. Anschließend wird der nächste Job aus der *joblist* entnommen und geprüft, ob dieser Job belegt werden soll. Algorithmus 2 zeigt diesen Vorgang in Form von Pseudo-Code.

In Zeile 1 wird geprüft, ob die Liste der Dienste bereits leer ist. Wenn noch ein Job belegt werden kann, wird der Job mit seiner Dienstgüte aus der *joblist* entnommen und entfernt (Zeile 2). Nun wird die Dienstgüte des Jobs neu berechnet (Zeile 3). Anschließend wird mit $job.isOpen()$ geprüft, ob der Dienst noch nicht belegt wurde oder ob der Dienst von diesem Knoten mit einer besseren Dienstgüte erbracht werden kann (Zeile 4). Trifft eines der beiden Ereignisse zu, wird der Dienst von dem Knoten belegt (Zeile 5). Die lokalen Ressourcen werden um den Ressourcenbedarf des Jobs verringert (Zeile 6) und die anderen Knoten werden über die Belegung informiert (Zeile 7).

Wie aus Algorithmus 2 zu erkennen ist, wird ein Dienst erst dann aus der *joblist* entfernt, wenn er von dem lokalen Knoten bearbeitet wird. Eine frühere Belegung wird bei dem betroffenen Job zwar in der *joblist* vermerkt, aber erst dann geprüft, wenn der Job auch tatsächlich für eine Belegung betrachtet wird. Ist zu diesem Zeitpunkt die Dienstgüte auf dem Knoten besser als die des bereits zugewiesenen Knotens, wird die vorhandene Belegung überschrieben.

Mit dieser Vorgehensweise und der Sortierung der Jobs anhand ihrer Dienstgüte auf den Knoten ist gewährleistet, dass jeder Job in der Reihenfolge betrachtet wird, in der er am besten von einem Knoten erbracht werden könnte. Das bedeutet, dass jeder Knoten zunächst versucht seine favorisierten Jobs zu belegen.

Algorithmus 2 Job Auswahl

```
1: while !empty(joblist) do
2:   (qos, job) ← joblist.removeFirst();
3:   quality ← calculateQualityOfService(job);
4:   if job.isOpen() or job.canBeProvidedBetter(quality) then
5:     provide(job);
6:     reduceAvailabeCapacity(job.getRequirements());
7:     notifyOthers(job, quality);
8:   end if
9: end while
```

Für den Fall, dass mehrere Instanzen eines Dienstes erbracht werden müssen, wird bei dem entsprechenden Job eine Liste aller Dienstanbieter (Knoten) hinterlegt. Die Liste der Dienstanbieter wird ebenfalls in absteigender Reihenfolge, abhängig von der Dienstgüte sortiert. Ist diese Liste bereits voll, wird bei der Prüfung $job.canBeProvidedBetter(quality)$ getestet, ob der Knoten den Dienst besser anbieten kann, wie einer der eingetragenen Knoten. Ist dies der Fall, wird der Knoten mit seiner Dienstgüte einsortiert und der Dienstanbieter mit der schlechtesten Dienstgüte wird am Ende der Liste entfernt. Da die Berechnung der Dienstgüte und die Sortierung anhand der bekannten Kriterien für alle Knoten eindeutig ist, entsteht auf allen anderen Knoten, die über die Überschreibung informiert werden die gleiche Sortierung der Dienstanbieter. Der Knoten, der aufgrund der neuen Belegung aus der Liste entfernt wurde, erkennt dies selbstständig. Da auch er zum gleichen Ergebnis kommt, erkennt er, dass er aus der Liste entfernt wird und kann die belegten Ressourcen wieder freigeben.

4.3.3 Konfliktlösung

Während der Belegungsphase der Jobs können die bereits aus dem Beispiel bekannten Probleme auftreten. Es kann vorkommen, dass zwei Knoten gleichzeitig eine Dienstbelegung bekannt geben, da kein Knoten weiß, wann ein anderer Knoten seine nächste Belegung bekannt geben wird.

Werden bei den gleichzeitig verschickten Belegungen unterschiedliche Jobs belegt, können beide Belegungen in die *joblist* eingetragen werden. Wird dabei der gleiche Job mit unterschiedlichen Dienstgüten belegt, wird nur der Knoten eingetragen, der die höhere Dienstgüte aufweist, es sei denn, dass für den Dienst noch mehrere Dienstanbieter benötigt werden. Dann können beide Knoten bei dem Job als Dienstanbieter eingetragen werden.

Kann jedoch nur einer der beiden Knoten bei dem Job eingetragen werden und beide Belegungen weisen zudem die gleiche Dienstgüte auf, entsteht ein Konflikt. Der Konflikt könnte durch Aushandlung zwischen den betroffenen Knoten aufgelöst werden, was jedoch bedeuten würde, dass zusätzliche Nachrichten benötigt werden. Aus diesem Grund wird eine Konfliktlösung benutzt, die es jedem Knoten ermöglicht einen Knoten eindeutig als Dienstanbieter auszuwählen.

Für die Konfliktlösung werden zusätzliche Informationen über die Knoten benötigt. Mit diesen Informationen kann über mehrere Stufen ein Knoten eindeutig ausgewählt werden. Die Konfliktlösung sieht die folgenden fünf Stufen vor. Kann der Konflikt nicht auf einer Stufe gelöst werden, weil die betrachteten Werte gleich sind, wird die nächste Stufe zur Entscheidungsfindung herangezogen.

1. *Last des Knotens*
   Der Job wird dem Knoten mit der geringsten Last zugewiesen.

2. *Anzahl der zugewiesenen Jobs*
   Der Knoten mit der geringsten Anzahl bereits zugewiesener Jobs wird als Dienstanbieter ausgewählt. Dabei wird die Annahme getroffen, dass mehr Jobs, von denen jeder in einem eigenen Thread läuft, auch eine höhere Belastung erzeugen, beispielsweise durch Thread-Wechsel.

3. *Anzahl noch belegbarer Jobs*
   Dem Knoten, der die geringste Anzahl noch belegbarer Jobs hat wird der Job zugewiesen, da die Wahrscheinlichkeit einen Dienst gut erbringen zu können bei einer kürzeren *joblist* schwerer sein wird, als mit einer größeren Anzahl an belegbarer Jobs.

4. *Zufallszahl*
   Konnte bis hierher noch keine Entscheidung getroffen werden, welcher Knoten den Dienst belegen soll, wird eine Zufallszahl benutzt, um zu verhindern, dass durch die folgende Stufe immer die gleiche Belegung erfolgt. Es ist jedoch sehr unwahrscheinlich, das die Zufallszahl zur Entscheidung herangezogen werden muss.

5. *Identifikation der Knoten*
   Da die Zufallszahl aus der vorherigen Stufe jeweils lokal auf jedem Knoten generiert wird, kann es theoretisch vorkommen, dass zwei Knoten die gleiche Zufallszahl besitzen. Um für das Problem trotzdem eine eindeutige Entscheidungsgrundlage zu bieten, wird letztendlich die Identifikation der Knoten herangezogen, die eindeutig sein muss, da jede Identifikation eines Knotens nur einmal im Netzwerk vorhanden sein kann.

Die Daten zur Konfliktlösung müssen bei jeder Belegung eines Jobs mit der Belegungsnachricht versendet werden, da es potentiell bei jeder Belegung auch zu einem Konflikt kommen kann. Bei den beschriebenen Daten handelt es sich jedoch lediglich um vier ganzzahlige Werte, die keinen großen Zusatzaufwand für den Nachrichtenaustausch bedeuten. Bereits eine zusätzliche Nachricht zur Lösung eines Konflikts würde mehr Aufwand bedeuten, abgesehen von der zeitlichen Verzögerung für das Versenden der Nachrichten. Der fünfte Wert, die Identifikation des Knotens, wird bereits mit der Belegungsnachricht versendet und ist ohnehin beim Empfänger verfügbar.

4.3.4 Verifikation der Konfiguration

Die zweite Phase der Selbstkonfiguration endet, wenn die *joblist* leer ist und somit alle Jobs belegt wurden (siehe Algorithmus 2). Um sicherzustellen, dass alle Knoten über die gleiche Belegung der Jobs verfügen, wird als dritte und abschließende Phase eine Verifikationsphase durchlaufen.

Der Knoten, bei dem alle Jobs als erstes vollständig als belegt gekennzeichnet sind, initiiert die Verifikationsphase. Nachdem er die letzte Belegung empfangen oder selbst versendet hat, wartet er eine gewisse Zeit, um mögliche Überschreibungen in die Liste einarbeiten zu können. Er verschickt eine Nachricht in der die Vollständigkeit der Konfiguration bekannt gegeben wird. In dieser Nachricht sendet der Knoten seine Konfiguration allen anderen Knoten zum Vergleich.

Im Beispiel zu Beginn dieses Kapitels hat der Projektleiter die Konfigurationsphase eröffnet. Im verteilten Netzwerk macht es keinen Sinn, auf einen ausgezeichneten Knoten zu warten, bis er die Verifikationsphase beginnt. Darum übernimmt diese Aufgabe der Knoten, der zuerst feststellt, dass er über eine vollständige Konfiguration verfügt.

Die Nachricht mit der vollständigen Konfiguration wird von den Knoten entgegengenommen und mit der lokalen Konfiguration verglichen. Dazu wird mit einem Hash-Verfahren ein Schlüssel berechnet, mit dem der Vergleich der Konfigurationen auf den Vergleich einfacher Zahlen zurückgeführt und damit effizient durchgeführt werden kann. Sind die Konfigurationen identisch, ist die Selbstkonfiguration abgeschlossen und die Applikation kann gestartet werden.

Unterscheiden sich die Konfigurationen voneinander, wird aus beiden Konfigurationen eine optimale Belegung berechnet. Besitzt der Empfänger der Nachricht eine schlechtere Belegung übernimmt er die empfangene Belegung, um seine Konfiguration zu verbessern. Besitzt jedoch der Empfängerknoten eine bessere Belegung, teilt er die optimierte Konfiguration allen anderen Knoten mit, damit auch diese ihre Konfiguration gegebenenfalls verbessern können.

Werden keine weiteren optimierten Konfigurationen versendet, ist die Selbstkonfiguration beendet und die Applikation kann gestartet werden, indem jeder Knoten seine Dienste und Monitore startet.

4.3.5 Unerfüllbare Konfigurationen

Die bisherigen Betrachtungen gingen von dem Fall einer erfüllbaren Konfiguration aus. Es kann jedoch durchaus vorkommen, dass eine Konfiguration aufgrund fehlender Ressourcen nicht erfüllt werden kann. Erfordert eine Konfiguration beispielsweise eine Datenbank, die jedoch keiner der Knoten im Netzwerk anbietet oder wird ein spezieller Sensor für die Applikation benötigt, der an keinem der Knoten verfügbar ist, wird der zugehörige Job bei allen Knoten in die Liste der *undoableJobs* eingetragen.

Eine Konfiguration ist erst erfüllt, wenn alle Jobs der Konfiguration durch einen Knoten belegt wurden, egal ob sie in der *joblist* oder in der *undoableJobs* Liste enthalten sind. Tritt der Fall ein, dass kein Knoten einen bestimmten Job

erbringen kann, endet die Belegung der Jobs, ohne dass diesem Job ein Knoten zugewiesen wurde.

Jeder Knoten geht davon aus, dass zwischen zwei Belegungsnachrichten eine maximale Zeitspanne nicht überschritten wird. Wird diese Zeitspanne überschritten und gibt es zudem noch nicht zugewiesene Jobs, geht der Knoten in einen Alarmzustand über. In diesem Zustand muss der Knoten klären, ob er vom Rest des Netzwerks getrennt wurde und somit nicht weiter an der Selbstkonfiguration mitarbeiten kann, oder ob der nicht zugewiesene Job tatsächlich nicht belegt werden kann und damit die Konfiguration unerfüllbar wäre.

Erkennt ein Knoten, dass die maximale Zeitspanne zwischen zwei Nachrichten verstrichen ist, verschickt er eine explizite Anfrage bezüglich des unbelegten Jobs an alle anderen Knoten. Belegt ein Knoten diesen Job antwortet er auf die Anfrage des Knotens. Dieser Fall tritt ein, wenn ein Knoten aus irgendwelchen Gründen die Belegungsnachricht des anderen Knotens nicht erhalten hat.

Belegt kein Knoten den Job antwortet auch keiner der anderer Knoten. Damit kann der Sender jedoch noch nicht entscheiden, ob er vom Netzwerk getrennt wurde oder ob der Dienst nicht erbracht werden kann. Der Knoten sendet daraufhin einen „Ping“ an die anderen Knoten. Auf diese Nachricht meldet sich auf jeden Fall ein Knoten, wenn er die Nachricht empfängt. Erhält der Knoten auf seinen „Ping“ eine Antwort, weiß er, dass er nicht vom Netzwerk getrennt wurde und dass somit die Konfigurationsanforderung nicht erfüllt werden kann.

Alle Knoten, die den „Ping“ empfangen haben, schließen daraus, dass die Konfigurationsanforderung nicht erfüllbar ist, da sonst der Knoten, der den Job belegt auf die explizite Anfrage des Knotens geantwortet hätte. Damit gehen die anderen Knoten ebenfalls in den Zustand „unerfüllbare Konfiguration“ über.

4.4 Evaluation

Die Evaluierung soll zeigen, dass bereits der Ansatz mit einer einfachen Metrik ohne zentrale Kontrolle zu guten Ergebnissen führt und auch in großen Netzwerken angewendet werden kann. Besonders für die Forderung der Anwendbarkeit in großen Netzen ist es wichtig, dass der Algorithmus ein möglichst lineares Wachstum in Bezug auf die versendeten Nachrichten aufweist.

Hierzu soll zunächst eine Komplexitätsbetrachtung Aufschluss darüber geben, wie viele Nachrichten der Algorithmus in unterschiedlichen Fällen benötigt. Anschließend wird die Simulationsumgebung beschrieben und die Evaluationsergebnisse werden präsentiert.

4.4.1 Komplexitätsbetrachtungen

Die Komplexitätsbetrachtung soll zeigen, wie viele Nachrichten im besten, im schlechtesten und in einem suboptimalen Fall für die Selbstkonfiguration benötigt werden. Für die folgenden Betrachtungen wird angenommen, dass n die Anzahl der vorhandenen Knoten im Netzwerk ist und j die Anzahl der Jobs der Konfigurationsanforderung.

Bester Fall

Der beste Fall tritt ein, wenn für die Belegung jedes Jobs genau eine Nachricht benötigt wird und für die Verifikationsphase ebenfalls eine Nachricht ausreicht, da alle Knoten die gleiche Belegung für die Konfiguration besitzen. Zusätzlich wird für die Verteilung der Konfigurationsanforderung eine Nachricht benötigt. Damit kann die untere Schranke des Algorithmus angegeben werden.

$$2 + j = o(j) \tag{4.6}$$

Die Anzahl der Nachrichten ist im besten Fall ausschließlich von der Anzahl der Jobs in der Konfigurationsanforderung abhängig. Damit skaliert der Algorithmus linear mit der Anzahl der zu belegenden Jobs und ist unabhängig von der Größe des Netzwerks.

Schlechtester Fall

Im schlechtesten Fall beginnt der Knoten mit der schlechtesten Dienstgüte einen Job zu belegen. Der Knoten mit der zweitschlechtesten Dienstgüte überschreibt die vorherige Belegung. Wird jeder Job in aufsteigender Reihenfolge der Dienstgüte von jedem Knoten einmal überschrieben, entstehen dadurch pro Job n Nachrichten.

Für die Verifikation kann im schlechtesten Fall das gleiche Szenario angenommen werden, bei dem eine Konfiguration durch den nächsten Knoten wieder verbessert werden kann, bis schließlich der letzte Knoten die endgültige Konfiguration berechnet. Hierbei würden auch n Nachrichten verschickt werden. Mit der benötigten Nachricht zum Verteilen der Konfiguration wäre der schlechteste Fall

$$1 + n^{j+1} = O(n^{j+1}) \tag{4.7}$$

Im schlechtesten Fall zeigt der Algorithmus ein exponentielles Wachstum in der Anzahl der Nachrichten. Damit wäre er für den Einsatz in großen Netzen ungeeignet.

Suboptimaler Fall

Es soll ein weiterer Fall betrachtet werden, der das reale Verhalten des Algorithmus besser beschreibt. Im suboptimalen Fall wird davon ausgegangen, dass es aufgrund von Überschreibungen in der Belegungsphase und durch Verbesserungen während der Verifikationsphase zu einem größeren Aufkommen an Nachrichten kommt, als im besten Fall. Allerdings wird in beiden Phasen nicht der schlechteste Fall eintreten, sondern im Schnitt wird jeder Knoten eine Überschreibung während der Zuweisungsphase oder eine Verbesserung in der Verifikationsphase durchführen. Dann berechnet sich die Anzahl der benötigten Nachrichten im suboptimalen Fall zu

$$2 + j + n = O(j + n) \tag{4.8}$$

In der Komplexitätsbetrachtung ergibt sich daraus wieder ein lineares Verhalten des Selbstkonfigurationsalgorithmus. Kann der suboptimale Fall erreicht werden, eignet sich der Algorithmus auch für Netzwerke mit einer großen Anzahl an Knoten.

4.4.2 Evaluationsmethodik

Zur Evaluation der Selbstkonfiguration wurde ein Simulator in Java implementiert mit dem es möglich ist, verschiedene Parameter der Simulation zu verändern. In Abbildung 4.1 ist das Hauptfenster des Simulators während der Simulation einer Selbstkonfiguration in einem Netzwerk mit 100 Knoten abgebildet.

Im linken Bild werden gerade die Jobs zugewiesen, während im rechten Bild einige der Knoten bereits eine vollständige Konfiguration besitzen. Jedes Element der Matrix repräsentiert einen Knoten des Netzwerks. Die blaue Farbe gibt Auskunft darüber, wie viele Jobs der Konfiguration bereits belegt sind. Je heller die Farbe ist, desto mehr Jobs sind bereits zugewiesen. Die grüne Farbe zeigt an, dass die Konfiguration auf diesem Knoten vollständig ist und die Konfiguration verifiziert werden kann. Am Ende einer erfolgreichen Konfiguration sind alle Knoten grün.

4.4.3 Evaluationsergebnisse

Pro Simulation können verschiedene Parameter variiert werden, um die Auswirkung bei veränderten Startbedingungen evaluieren zu können. Als Parameter können die Anzahl der Knoten im Netzwerk, die prozentuale Auslastung der Knoten und die Anzahl unterschiedlicher Ressourcen variiert werden. Darüber hinaus kann zwischen einer heterogenen und einer homogenen Hardware unterschieden werden.

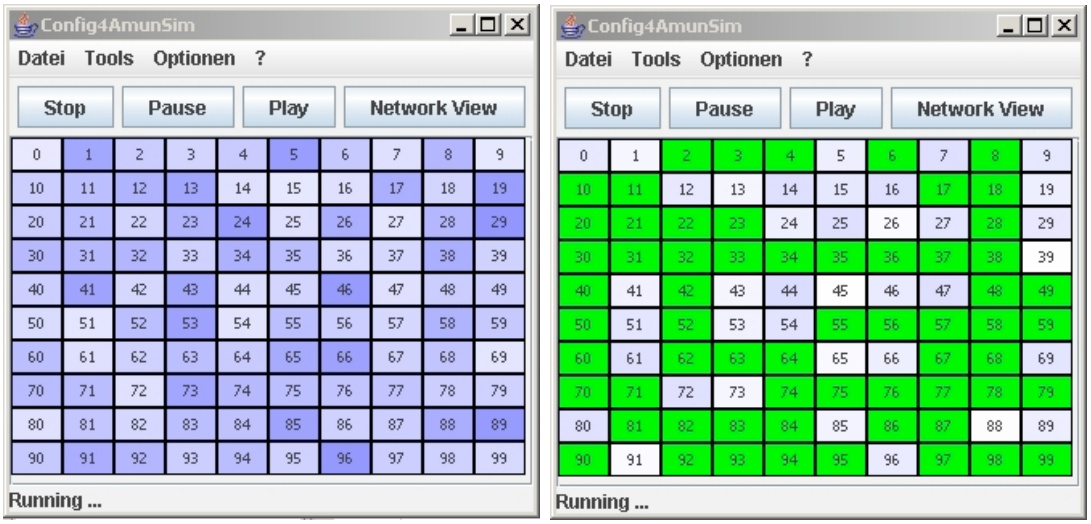

Abbildung 4.1: Simulator während einer Selbstkonfiguration eines Netzes mit 100 Knoten

Im Fall einer homogenen Hardware wird ein Netzwerk mit identischen Knoten erzeugt, in dem jeder Knoten die gleichen Ressourcen mit den gleichen Kapazitäten je Ressource besitzt. Eine homogene Hardware ist sehr häufig in Sensornetzwerken zu finden oder in Anwendungen des Grid- beziehungsweise Cluster-Computing.

Bei heterogener Hardware werden die Kapazitäten der vorhandenen Ressourcen um einen prozentualen Anteil variiert. Es wird ein Netzwerk mit Knoten generiert, bei denen sich die Ressourcen unterscheiden. Heterogene Hardware wird häufig in gewachsenen Umgebungen vorgefunden, bei denen im Laufe der Zeit unterschiedliche Computer in ein Netzwerk integriert wurden. Ubiquitäre Systeme sind ebenfalls ein Beispiel für eine heterogene Hardwarelandschaft, da hier sehr viele Geräte mit teils stark unterschiedlichen Hardwarevoraussetzungen in einem Netzwerk interagieren.

Jede Parameterkombination wurde 100-mal simuliert. Für jede der 100 Simulationen wurde anhand der Parametereinstellungen eine Hardwarevorgabe für die Knoten und eine Konfigurationsanforderung generiert. Mit dieser Vorgehensweise ist sichergestellt, dass nicht zufällig ein Fall gewählt wird, der besonders gute Ergebnisse liefert. Die Streuung der Werte durch die zufällige Generierung der Anfangsbedingungen im Rahmen der Vorgaben, ermöglicht es zuverlässige Aussagen über die Robustheit der Selbstkonfiguration zu treffen.

Abbildung 4.2 zeigt die Anzahl der Nachrichten bei 100 Simulationsläufen. Anhand der Anzahl der Nachrichten für den besten Fall ist ersichtlich, wie stark die initialen Werte voneinander abweichen, da im besten Fall die Anzahl der Nachrichten proportional zur Anzahl der Jobs der Konfiguration ist.

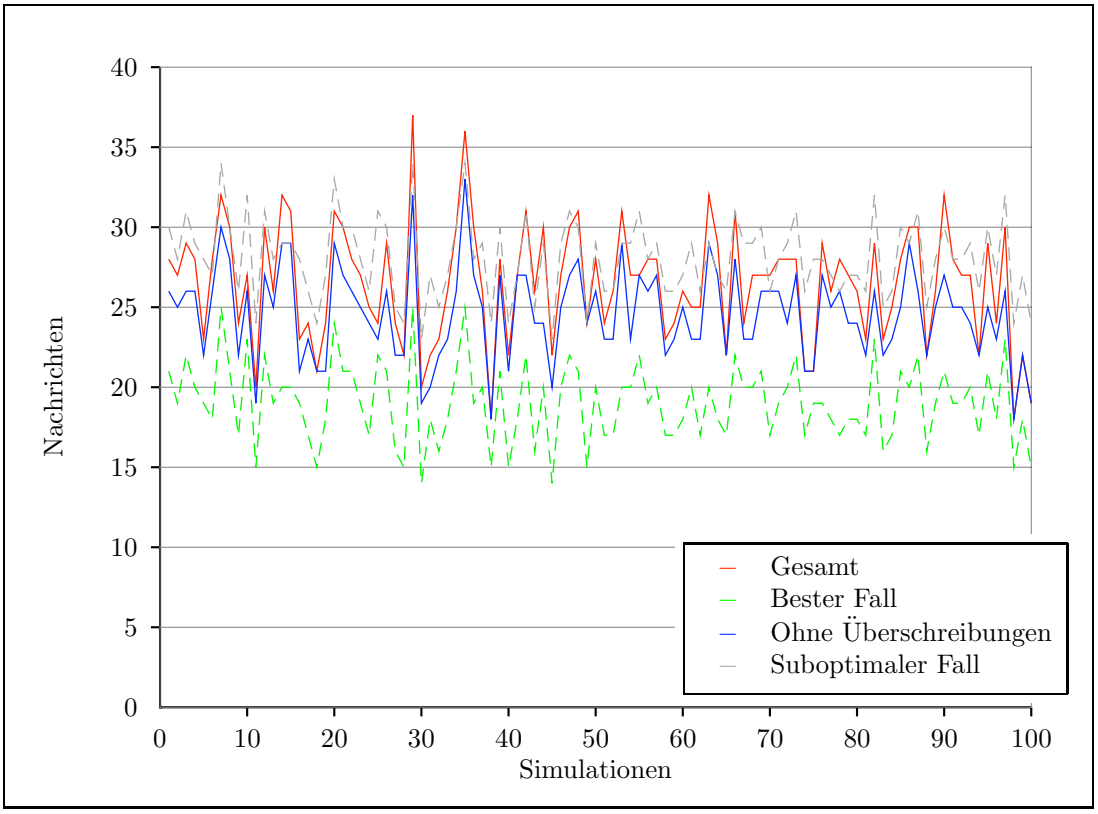

Abbildung 4.2: Beispiel einer Simulation mit 100 Simulationsläufen

Variation des Ressourcenverbrauchs

Die prozentuale Auslastung des Netzes wird anhand der vorhandenen Ressourcen berechnet. Soll die Konfigurationsanforderung die Knoten zu 50 Prozent auslasten, werden dazu die Werte der Ressource aller Knoten addiert und dann so viele Jobs generiert, bis 50 Prozent der Ressourcen durch die Konfigurationsanforderung verbraucht werden. Dabei wird darauf geachtet, dass kein Job so viele Ressourcen verbraucht, dass er von keinem der Knoten mehr erbracht werden könnte. Damit werden grundsätzlich erfüllbare Konfigurationsanforderungen erstellt.

In Abbildung 4.3 ist die Anzahl der benötigten Nachrichten für homogene Hardware auf der linken und für heterogene Hardware auf der rechten Seite abgebildet. Die Diagramme zeigen die Ergebnisse für 10, 25, 50 und 100 Knoten bei unterschiedlichen Auslastungen von 20, 60, 80 und 100 Prozent, wobei drei Ressourcen benutzt wurden.

Die Simulationen bei homogener Hardware zeigen bei allen Netzgrößen ein sehr gutes Verhalten in Bezug auf die Anzahl der notwendigen Nachrichten. Die Werte

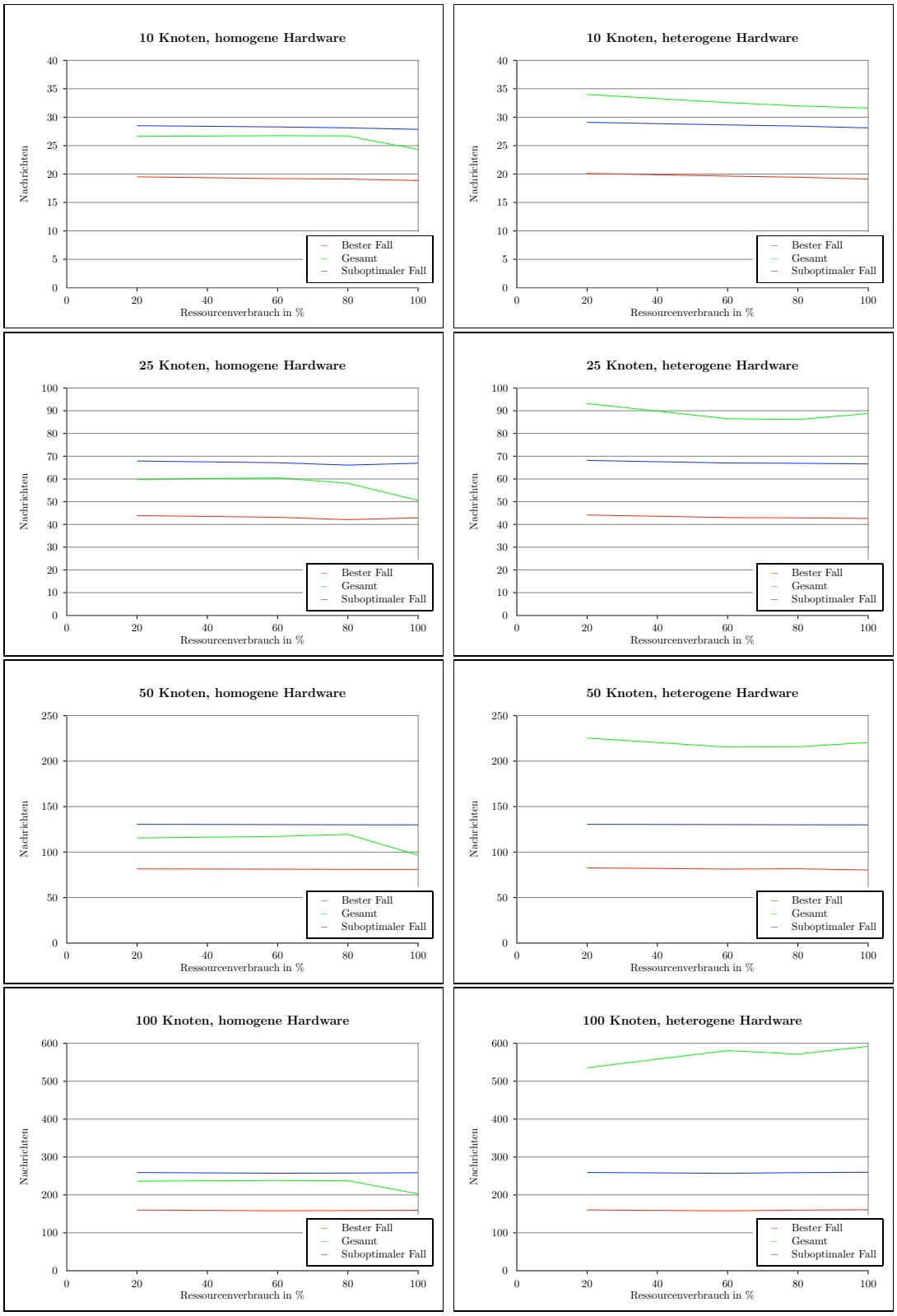

Abbildung 4.3: Anzahl der Nachrichten bei unterschiedlicher Auslastung der Ressourcen bei homogener und heterogener Hardware

bewegen sich immer zwischen dem besten und dem suboptimalen Fall. Darüber hinaus ist zu erkennen, dass die prozentuale Auslastung fast keinen Einfluss auf die Anzahl der Nachrichten hat. Bei 100 Prozent wurde bei allen Simulationen eine geringere Anzahl an Nachrichten für die Selbstkonfiguration benötigt, was darauf zurückgeführt werden kann, dass in diesem Fall durch die vollständige Auslastung der Ressourcen weniger Überschreibungen stattfinden als in den anderen Fällen.

Bei heterogener Hardware sind die Ergebnisse deutlich schlechter. Es werden grundsätzlich mehr Nachrichten wie im suboptimalen Fall benötigt. Mit zunehmender Netzgröße verschlechtert sich das Ergebnis weiter.

Das Problem bei heterogener Hardware ist die stark unterschiedliche Bewertung der Jobs anhand der vorhandenen Ressourcen. Bei homogener Hardware werden die Jobs gerade zu Beginn der Selbstkonfiguration auf allen Knoten gleich bewertet. Daraus ergibt sich auf allen Knoten die gleiche Sortierung der *joblist*. Wird ein Job belegt, können die anderen Knoten den Job nicht mit einer besseren Dienstgüte erbringen. Damit wird eine große Anzahl an Überschreibungen verhindert.

Im heterogenen Fall tritt genau das Problem der vielen Überschreibungen zu Tage. Da kein Knoten die Dienstgüten der anderen Knoten kennt, kann er auch nicht wissen, wann er eine Belegung bekannt geben soll. Im optimalen Fall würde ein Knoten so lange warten, bis er erkannt hat, dass seine berechnete Dienstgüte in etwa der entspricht, die bei der letzten Zuweisung verschickt wurde. Damit werden Überschreibungen unwahrscheinlicher und die Anzahl der Nachrichten würde sich reduzieren.

Variation der Anzahl der Ressourcen

Die vorherigen Simulationsergebnisse wurden mit drei Ressourcen durchgeführt. Je mehr Ressourcen für die Generierung der Konfigurationsanforderung zur Verfügung stehen, desto unterschiedlicher können die generierten Jobs ausfallen, da auch die Anzahl der benutzten Ressourcen zufällig ermittelt wird. Da alle benutzten Ressourcen in die Berechnung der Dienstgüte eingehen, verändert sich gerade im homogenen Fall die Auslastung der einzelnen Knoten sehr schnell.

Die Simulationen in Abbildung 4.4 zeigen das Verhalten der Selbstkonfiguration bei steigender Anzahl unterschiedlicher Ressourcen. Es wurde eine Auslastung von 80 Prozent gewählt. Die Diagramme zeigen, dass für alle Größen des Netzwerks bei mehr Ressourcen eine größere Anzahl an Nachrichten benötigt werden. Bereits ab fünf Ressourcen liegt die Anzahl der Nachrichten auch für den Fall einer homogenen Hardware über dem suboptimalen Fall.

Die Anzahl der Nachrichten im homogenen Fall wächst bei unterschiedlichen Netzgrößen in etwa gleich stark. Bei heterogener Hardware fällt der Anstieg mit

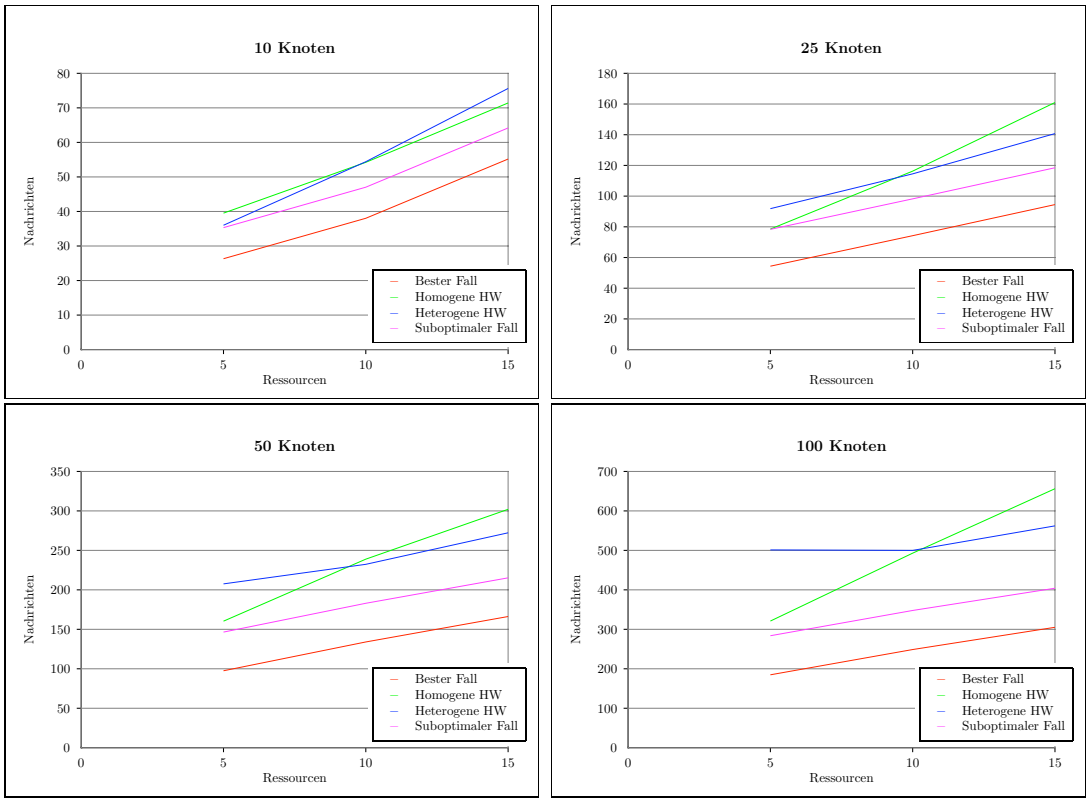

Abbildung 4.4: Nachrichten bei unterschiedlicher Ressourcenanzahl

zunehmender Knotenzahl geringer aus. In allen Simulationen ist zu erkennen, dass bei steigender Zahl verschiedener Ressourcen die Selbstkonfiguration im Fall einer heterogenen Hardware deutlich weniger Nachrichten benötigt als im Fall der homogenen Hardware.

Im heterogenen Fall ist die Anzahl der Nachrichten zwar über dem suboptimalen Fall, verläuft aber annähernd parallel dazu. Dies lässt den Schluss zu, dass der suboptimale Fall eine gute Abschätzung für das Nachrichtenaufkommen darstellt und dass das lineare Verhalten durch die Simulationen bestätigt werden kann. Lediglich die Größe des konstanten Faktors, der in der Komplexitätsbetrachtung eliminiert werden kann, muss angepasst werden. Grundsätzlich kann bei der Anzahl der Nachrichten von einer Abhängigkeit von der Anzahl der Jobs in der Konfiguration ausgegangen werden, da mit zunehmender Größe des Netzes bei gleich bleibender Auslastung (beispielsweise 80 Prozent) auch mehr Jobs generiert werden.

Mittlere Nachrichtenanzahl

Die Anzahl der Nachrichten, die zur Selbstkonfiguration benötigt werden ist ein Maß für die Güte des Algorithmus. In Abbildung 4.5 sind die Nachrichten pro Job für homogene und heterogene Hardware, bei 20, 60, 80 und 100 Prozent Ressourcenauslastung für verschiedene Netzgrößen dargestellt.

Bei homogener Hardware ist bis 80 Prozent ein leichter Anstieg der Nachrichten pro Job zu erkennen, wobei die Werte zwischen 1,4 und 1,6 schwanken. Das bedeutet, dass pro Job ungefähr 1,5 Nachrichten generiert werden und somit etwa die Hälfte der Jobs einmal überschrieben werden. Wird das Netzwerk zu 100 Prozent ausgelastet scheint die Zuweisung der Jobs eindeutiger zu sein, wodurch eine deutlich geringere Anzahl an Überschreibungen stattfindet und somit insgesamt weniger Nachrichten erzeugt werden.

Im Fall der heterogenen Hardware ist mit zunehmender Netzgröße eine Steigerung in der Anzahl der Nachrichten pro Job zu erkennen. Dafür schwanken die Werte bei unterschiedlichen Auslastungen der Netze deutlich weniger.

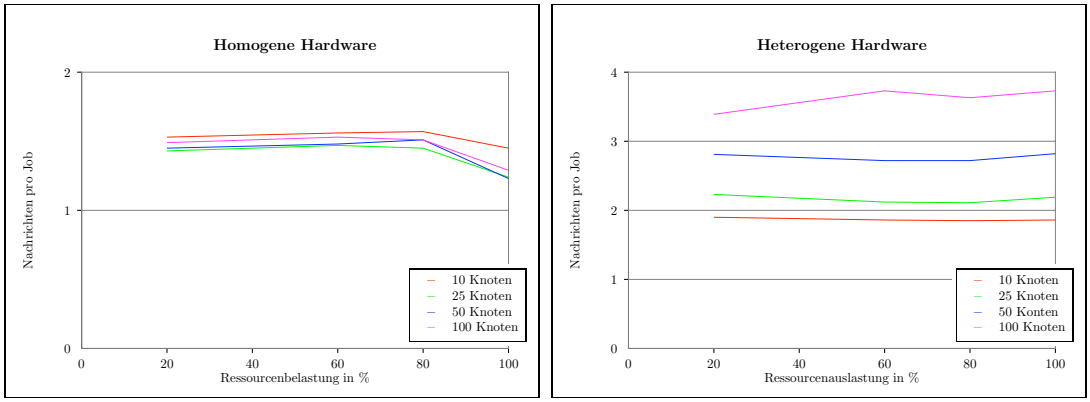

Abbildung 4.5: Mittlere Anzahl an Nachrichten bei homogener und heterogener Hardware

4.5 Verbesserungen

Die Evaluationen zeigen, dass der Selbstkonfigurationsalgorithmus bereits ein sehr gutes Verhalten selbst in großen Netzen zeigt. Lediglich im Fall von heterogener Hardware besteht Optimierungsbedarf, um die Anzahl der Nachrichten zu senken.

Ein wesentlicher Punkt im Ablauf der Selbstkonfiguration ist die Bekanntgabe einer Belegung. Da es keine zentrale Kontrolle gibt, kann kein Knoten wissen,

wann die nächste Belegung durch einen Knoten verkündet wird. Zudem ist die Qualität der nächsten Belegung ebenfalls unbekannt. Um jedoch die Anzahl der Überschreibungen zu reduzieren, müsste sowohl der Zeitpunkt, als auch die Qualität der nächsten Belegungsnachricht bekannt sein.

Wüsste ein Knoten, dass er die beste Dienstgüte für einen Dienst besitzt, könnte er sofort eine Belegung bekannt geben und es würde zu keiner Überschreibung kommen, da kein anderer Knoten eine bessere Dienstgüte für diesen Job aufweisen kann.

Aus der Dienstgüte kann somit abgeleitet werden, wann ein Knoten eine Belegung bekannt geben muss. Da jeder Knoten zwischen einem passiven Zustand, in dem er nur auf eingehende Nachrichten wartet und einem aktiven Zustand, in dem er einen Job belegen kann, wechselt, kann die Zeitspanne, die der Dienst bis zur nächsten Belegung wartet, anhand der letzten empfangenen Dienstgüte berechnet werden.

Jeder Dienst verweilt eine definierte Zeit im passiven Zustand und wechselt dann in den aktiven Zustand. Diese Zeitspanne t_{wait}wird mit dem Verhältnis aus der letzten empfangenen Dienstgüte qos_{net} und der besten eigenen Dienstgüte qos_{own} multipliziert.

$$t_{sleep} = \frac{qos_{net}}{qos_{own}} t_{wait} \tag{4.9}$$

Ist die eigene Dienstgüte kleiner als die letzte empfangene Dienstgüte, wird der Quotient aus qos_{net} und qos_{own} größer 1 und die Wartezeit verlängert sich. Ist die eigene Dienstgüte hingegen größer, wird der Bruch kleiner 1 und die Wartezeit verkürzt sich.

Damit ist gewährleistet, dass Knoten, die eine höhere Dienstgüte aufweisen, diese auch früher bekannt geben, als Knoten mit einer geringeren Dienstgüte und somit ein Teil der Überschreibungen eliminiert werden kann.

4.6 Integration in die Middleware

Die Integration der Selbstkonfiguration in die Middleware erfordert nur wenige zusätzliche Klassen, da bereits bei der Implementierung der Simulationsumgebung darauf geachtet wurde, dass die Struktur und damit auch die Abläufe innerhalb der Middleware so genau wie möglich nachgebildet werden.

4.6.1 Komponenten der Selbstkonfiguration

Die Komponenten der Selbstkonfiguration wurden so implementiert, dass die Integration möglichst einfach erfolgen kann. Die Schnittstellen zur Middlewa-

re wurden weitestgehend beibehalten, was den Wechsel von der Simulation zur Middleware sehr einfach gestaltet. Insgesamt wurden 40 Klassen aus der Simulationsumgebung der Selbstkonfiguration übernommen, fünf neue Klassen wurden implementiert, um die Selbstkonfigurationskomponenten zu integrieren und eine Klasse zur Steuerung der Evaluation.

Bei den Klassen aus der Simulationsumgebung sind zwei Generatoren enthalten mit denen die Hardware der einzelnen Knoten und die Konfigurationen für die Simulationen generiert werden können. Die Generatoren werden sowohl für die Simulation, wie auch für die Evaluation in der Middleware benutzt, um die Ergebnisse bei gleichen Voraussetzungen vergleichen zu können.

An dieser Stelle sollen nur die wichtigsten Klassen der Selbstkonfiguration beschrieben werden, um einen Überblick über die, für das Verständnis wichtigen Komponenten, zu geben. Abbildung 4.6 zeigt die Klassen der Selbstkonfiguration.

Eine Konfigurationsanforderung kann mit dem `ConfigLoader` geladen werden. Die XML-Datei mit der Konfigurationsanforderung wird in ein `Configuration`-Objekt umgesetzt, das eine Liste mit `Jobs` und `Constraints` enthält. Die Klasse `Jobs` ist die Oberklasse zu den verwendeten Objekten `Service` und `Monitor`.

Sowohl `Jobs` wie auch `Constraints` besitzen Elemente des Typs `Resource`. Bei den `Constraints` beschreibt eine `Resource` die notwendigen Voraussetzungen, damit eine Bedingung erfüllt ist. Bei einem `Job` hingegen wird durch eine

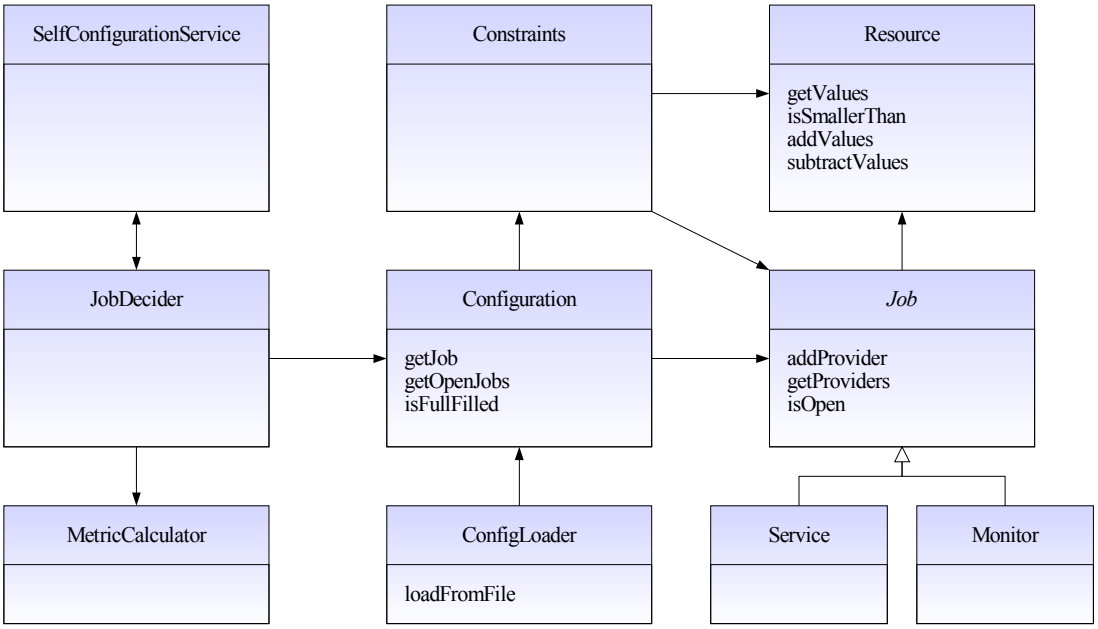

Abbildung 4.6: Klassendiagramm der Selbstkonfigurationskomponenten

`Resource` der tatsächliche Verbrauch dieser Ressource beschrieben, der für die Berechnung der Dienstgüte herangezogen wird.

Die Klasse `SelfConfigurationService` stellt den Dienst zur Steuerung der Selbstkonfiguration dar und implementiert die Schnittstelle `JobDispatcher`. Der `SelfConfigurationService` empfängt alle Nachrichten bezüglich der Selbstkonfiguration und leitet die Inhalte der Nachrichten an den `JobDecider` weiter. Der `JobDecider` verarbeitet die eingehenden Belegungen und berechnet mit dem `MetricCalculator` die einzelnen Dienstgüten, um zu entscheiden, welcher Dienst als nächstes belegt werden soll.

Über die Schnittstelle `JobDispatcher` informiert der `JobDecider` den `SelfConfigurationService` über eine getroffene Entscheidung, wie beispielsweise eine Dienstbelegung, die vollständige Belegung der Konfiguration oder die Erkennung eines Konfigurationskonflikts während der Verifikationsphase.

4.6.2 Komponenten für die Evaluation der Selbstkonfiguration

An dieser Stelle soll der Ablauf der Selbstkonfiguration im Zusammenspiel der Komponenten erklärt werden. Abbildung 4.7 zeigt einen Überblick über die Klassen, die maßgeblich den Ablauf der Selbstkonfiguration steuern.

Zu Beginn der Selbstkonfiguration wird die Konfigurationsspezifikation in Form einer XML-Datei durch den `ConfigLoader` in ein entsprechendes `Configuration`-Objekt umgesetzt. Die Konfiguration wird anschließend an alle Knoten durch den `SelfConfigurationService` verteilt. Der `SelfConfigurationService` ist für die Kommunikation zwischen den Knoten zuständig. Hat ein `SelfConfigurationService` eine Konfiguration empfangen, übergibt er diese an den `JobDecider`.

Der `JobDecider` prüft zunächst die enthaltenen `Constraints` und legt alle nicht erfüllbaren Jobs in die *undoableJobs*-Liste. Anschließend bewertet er die übrigen Dienste und sortiert sie absteigend nach ihrer Dienstgüte. Der `SelfConfigurationService` wechselt nun in den passiven Zustand und wartet auf eingehende Nachrichten.

Wechselt der `SelfConfigurationService` nach einer Wartezeit in den aktiven Zustand, wählt der `JobDecider` den nächsten zu erbringenden Dienst aus, der vom `SelfConfigurationService` allen anderen Knoten mitgeteilt wird. Die bereits eingegangenen Belegungen werden immer direkt verarbeitet und in die *jobList* eingetragen. Danach wartet der `SelfConfigurationService` so lange, bis seine Wartezeit für den Zeitraum des passiven Zustands vergangen ist, bevor er in den aktiven Zustand wechselt.

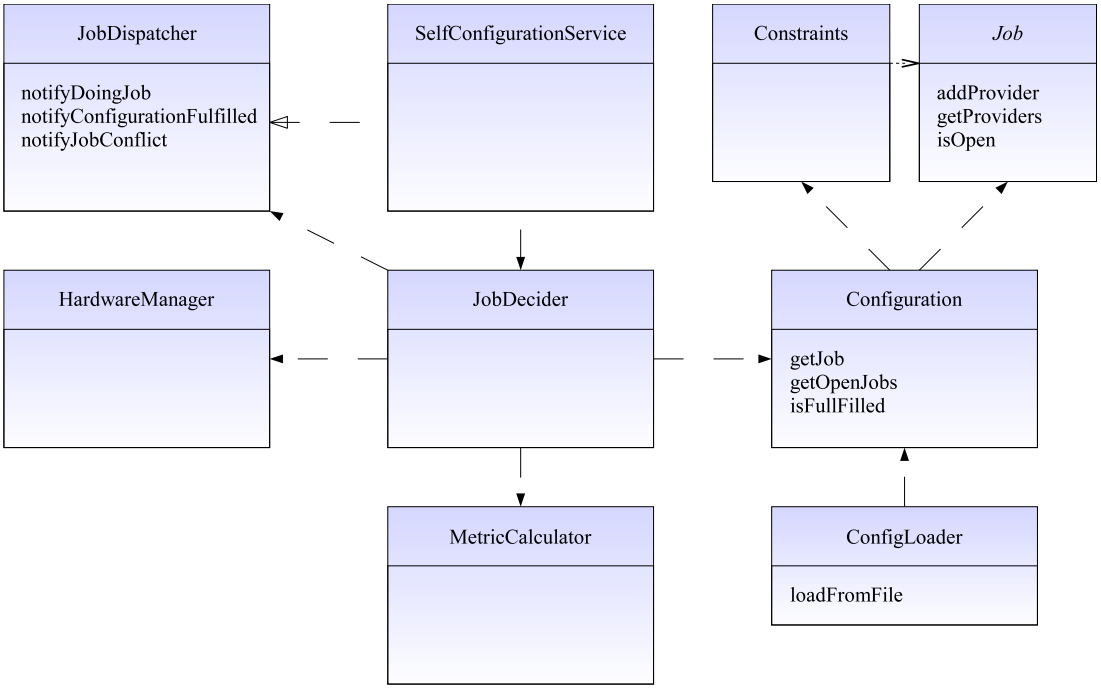

Abbildung 4.7: Klassendiagramm einiger Selbstkonfigurationskomponenten

Wird ein `Job` belegt, müssen die benötigten Ressourcen von den noch freien Ressourcen abgezogen werden, damit die Dienstgüte des nächsten Dienstes korrekt berechnet werden kann. Der `HardwareManager` verwaltet die Ressourcen des lokalen Knotens und übernimmt die Berechnung der noch freien Kapazitäten. Sowohl bei einer Belegung, als auch bei einer Überschreibung wird der entsprechende Dienst dem `HardwareManager` übergeben, damit er die erforderlichen Ressourcen belegen, beziehungsweise die frei werdenden Ressourcen wieder hinzufügen kann.

4.6.3 Evaluation

Die Evaluation der Selbstkonfiguration in der Middleware wird, soweit möglich, mit den gleichen Einstellungen wie die Simulationen durchgeführt, um eine Vergleichbarkeit der Ergebnisse zu gewährleisten. Einziger beschränkender Faktor ist die Anzahl der verfügbaren Knoten. Da nur eine begrenzte Anzahl an Computern für die Evaluationen zur Verfügung standen, wurde die Selbstkonfiguration mit 10 Knoten durchgeführt.

Für die Evaluationen wurde der Generator der Simulationsumgebung zur Erzeugung der Konfigurationsanforderungen benutzt. Damit ist sichergestellt, dass bei gleichen Einstellungen, gleiche Voraussetzungen für die Evaluation in der Middleware vorhanden sind. Jede Evaluation wurde für jede Parametereinstel-

lung 100-mal durchgeführt, um die Ergebnisse mit denen aus den Simulationen vergleichen zu können und um Schwankungen und Ausreißer bei den Messungen zu eliminieren.

Variation des Ressourcenverbrauchs

Mit dem prozentualen Ressourcenverbrauch wird die maximale Auslastung der Knoten bestimmt. Die folgenden Evaluationen wurden jeweils mit 20, 60, 80 und 100 Prozent Auslastung der verfügbaren Ressourcen durchgeführt. Jede Testreihe wurde weiterhin mit drei, fünf, zehn und fünfzehn Ressourcen für den Fall einer homogenen und einer heterogenen Hardware evaluiert. Die Ergebnisse sind in Abbildung 4.8 abgebildet.

Bei allen Diagrammen fällt auf, dass für jede Anzahl von Ressourcen ein ähnliches Verhalten zu erkennen ist. Bei homogener Hardware ist bei 80 Prozent Ressourcenauslastung ein deutlicher Anstieg der Nachrichten zu erkennen, während bei heterogener Hardware ein deutlicher Anstieg bei 100 Prozent Ressourcenauslastung entsteht. Es gibt keine eindeutige Erklärung für diesen Anstieg. In den Evaluationen ist jedoch zu erkennen, dass bei diesen beiden Punkten ein deutlicher Anstieg in der Anzahl der Überschreibungen festgestellt werden kann.

Da alle Evaluationen 100-mal durchgeführt wurden, kann es sich dabei nicht um einzelne Ausreißer handeln die das Ergebnis verfälschen. Ein verändertes Verhalten beim Versand der Belegungsnachrichten kann ebenfalls ausgeschlossen werden. Die einzige Erklärung für das Verhalten kann in der Generierung der Jobs liegen. Wie bei den Simulationen, bei denen ebenfalls bei 80 Prozent ein Anstieg in der Nachrichtenzahl zu erkennen ist, könnte es sein, dass auch hier aus dem gleichen Grund viele Überschreibungen erzeugt werden.

Bei heterogener Hardware ist bei 100 Prozent Auslastung in den Simulationen lediglich ein geringer Anstieg der Nachrichtenzahl zu erkennen. Offensichtlich scheint es bei der Belegung der Knoten mit zunehmender Auslastung mehr Überschreibungen zu geben. Dies liegt einerseits daran, dass insgesamt mehr Dienste generiert werden und somit die Wahrscheinlichkeit von Überschreibungen ansteigt, andererseits an der einfachen Strategie, eine Belegung auch dann zu überschreiben, wenn die Dienstgüte nur geringfügig besser ist. Besonders im heterogenen Fall scheint sich dieses Verhalten stärker bemerkbar zu machen als bei homogener Hardware.

Positiv zeigt sich das Gesamtverhalten der Selbstkonfiguration, das sowohl bei homogener, wie auch bei heterogener Hardware eine etwa gleich große Anzahl an Nachrichten für die Verteilung der Dienste erfordert. In den Simulationen wurde besonders im heterogenen Fall ein deutlicher Anstieg der Nachrichtenanzahl gegenüber dem homogenen Fall verzeichnet. Lediglich bei 100 Prozent Ressourcenverbrauch werden im heterogenen Fall zu viele Nachrichten benötigt. Anson-

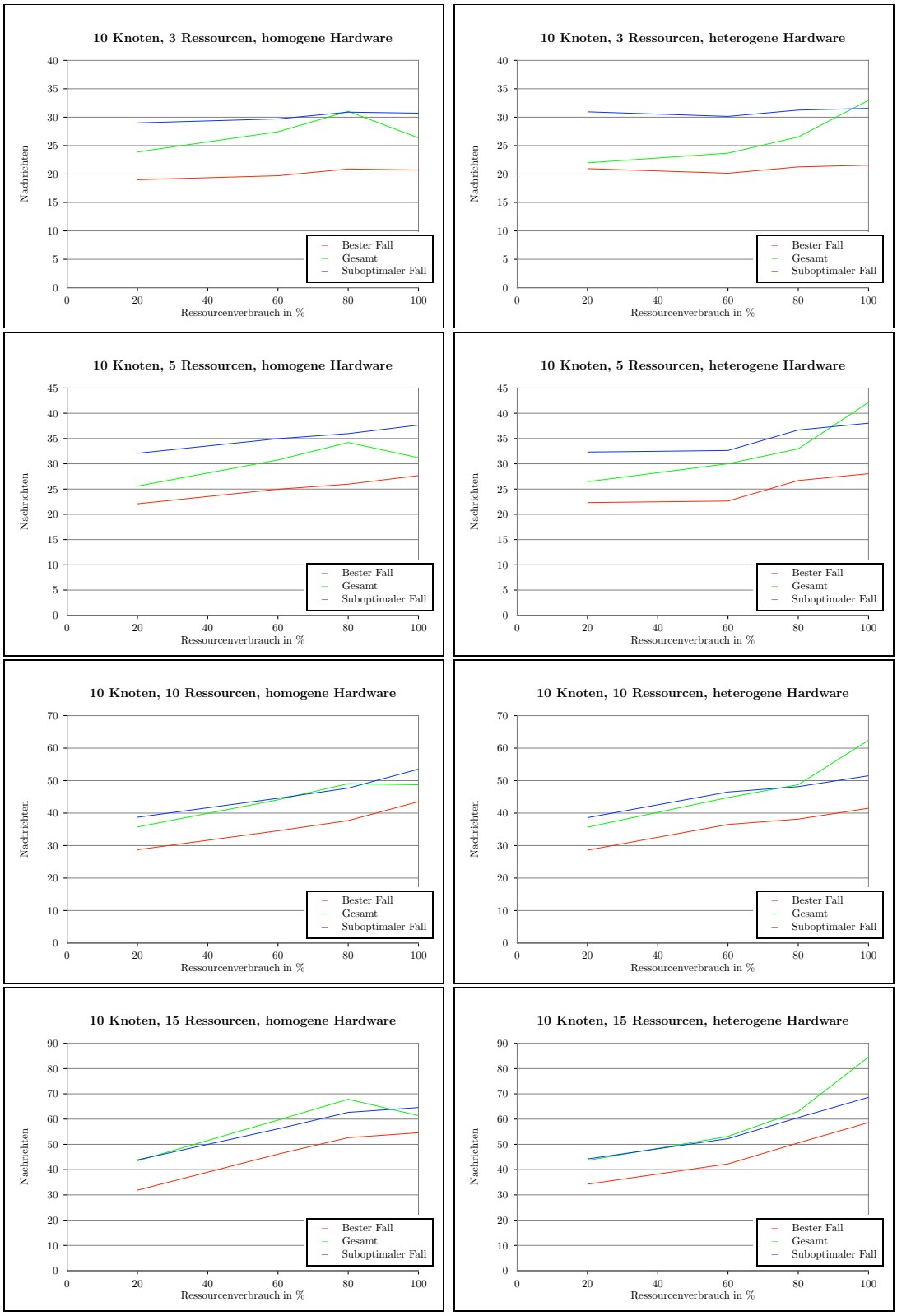

Abbildung 4.8: Anzahl der Nachrichten bei unterschiedlicher Auslastung der Ressourcen bei homogener und heterogener Hardware

sten bewegt sich die Nachrichtenanzahl immer zwischen dem optimalen und dem suboptimalen Fall.

Variation der Ressourcenzahl

Die Variation der Anzahl der Ressourcen zeigt das Verhalten der Selbstkonfiguration bei einer unterschiedlichen Anzahl an Parametern, die in die Berechnung der Dienstgüte eingehen. Die Ergebnisse sind in Abbildung 4.9 dargestellt. Der Vergleich mit den Simulationen aus Abbildung 4.4 zeigt, dass die Evaluation in der Middleware etwas bessere Ergebnisse erzielt.

Das Verhalten aus den Simulationen kann durch die Evaluationen in der Middleware bestätigt werden. Bei wenigen Ressourcen werden bei homogener Hardware etwas weniger Nachrichten benötigt als bei heterogener Hardware. Mit steigender Anzahl zu betrachtender Ressourcen werden bei heterogener Hardware weniger Nachrichten benötigt.

Zusätzlich werden sowohl bei homogener, wie auch bei heterogener Hardware insgesamt weniger Nachrichten benötigt als bei den Simulationen.

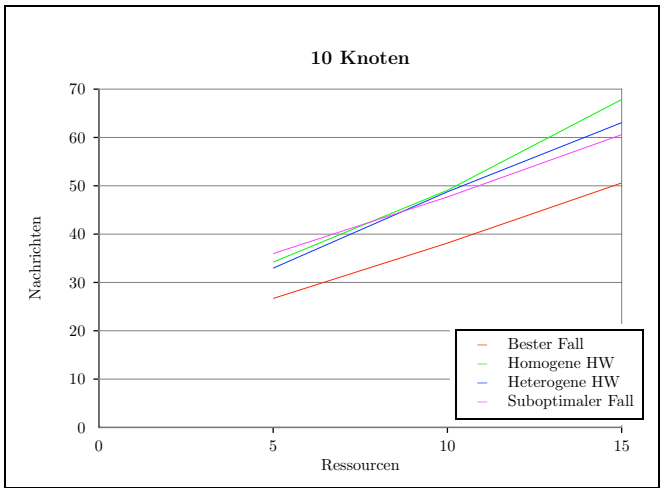

Abbildung 4.9: Anzahl der Nachrichten bei unterschiedlicher Ressourcenzahl

Mittlere Nachrichtenanzahl

Eine wichtige Maßzahl für die Beurteilung der Selbstkonfiguration ist die Anzahl der Nachrichten pro Job. In Abbildung 4.10 sind die Evaluationsergebnisse für unterschiedliche Ressourcenzahlen für homogene und heterogene Hardware bei 20, 60, 80 und 100 Prozent Auslastung dargestellt.

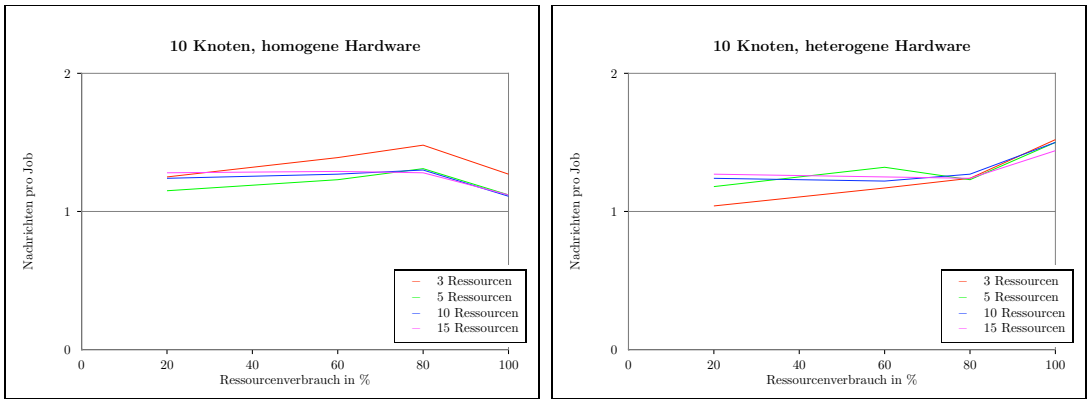

Abbildung 4.10: Mittlere Anzahl an Nachrichten bei homogener und heterogener Hardware

Die Werte der Evaluationen in der Middleware liegen sowohl bei homogener, als auch bei heterogener Hardware sehr nah beieinander. Dies lässt den Schluss zu, dass die Selbstkonfiguration in beiden Fällen sehr gute Ergebnisse liefert, da sich die Werte im Bereich zwischen 1,1 und 1,6 Nachrichten pro Job bewegen.

Die Ergebnisse der Simulationen liegen im Mittel etwa um 0,2 höher, was auf die Implementierung der Simulationsumgebung mit einem Thread pro Knoten zurückzuführen ist. Besonders bei einer großen Anzahl an Knoten wird ein sehr leistungsstarker Computer benötigt, um den Knoten die Möglichkeit zu geben die eingegangenen Nachrichten rechtzeitig zu verarbeiten und somit den Versand der Nachrichten mit einem zeitlich korrekten Verhalten durchführen zu können.

Es ist davon auszugehen, dass die Evaluationsergebnisse in der Middleware gerade aus diesem Grund etwas besser sind, da alle Knoten parallel arbeiten und die eingehenden Nachrichten sofort verarbeiten können. Damit ist das zeitliche Verhalten beim Versenden der Nachrichten weitestgehend unbeeinflusst von der Anzahl der versendeten Nachrichten, wodurch es zu etwas weniger Überschreibungen kommt.

4.7 Verwandte Forschungsarbeiten

Verschiedene Ansätze zur kooperativen Problemlösung und zur Selbstkonfiguration sind bereits in anderen Forschungsbereichen zu finden. Nachfolgend werden aus dem Bereich der Agententechnologie, einem theoretischen Ansatz zur bedingten Problemlösung und aus dem Bereich der Middleware, ähnliche Arbeiten erläutert.

4.7.1 Multi-Agenten-Systeme

Die Art der kooperativen Problemlösung, wie sie von der Selbstkonfiguration umgesetzt wird, erinnert an Multi-Agenten-Systeme [87, 12, 90]. Multi-Agenten-Systeme gehen im Allgemeinen von einer Anzahl verschiedener Agenten aus, die in kooperativer Art und Weise zur Lösung eines Problems zusammenarbeiten. Agenten werden in vielen Anwendungsfeldern eingesetzt, unter anderem zur Lösung logistischer Probleme. Der Vorteil von Agenten liegt in der Fähigkeit Planungen für komplexe Probleme verteilt durchführen zu können.

In der Agent Technologie Roadmap [35] werden verschiedene Anwendungsgebiete skizziert, in denen Agententechnologie vielversprechende Ansätze liefern. Es werden auch Beispiele genannt, in denen Lösungen auf Basis von Agentensystemen erarbeitet wurden. Für die Kommunikation zwischen den Agenten kommt die Agent Communication Language zum Einsatz, mit der ein applikationsunabhängiger Informationsaustausch realisiert werden kann.

Zentraler Punkt aller Agentensysteme ist die Verhandlung zwischen den beteiligten Agenten. Bereits 1980 gab es erste Bemühungen den Verhandlungsablauf zwischen Agenten zu normieren. In dem Artikel The Contract Net Protocol [62] wird ein Vorschlag zur Vereinheitlichung der Verhandlung zwischen den Agenten vorgeschlagen. Seit 1996 bemüht sich The Foundation for Intelligent Physical Agents (FIPA) [26] um die Erstellung einheitlicher Standards im Bereich der Agententechnologie.

Die Verhandlung zwischen Agenten läuft grundsätzlich nach einem Schema, bei dem ein Agent eine Offerte an andere Agenten verbreitet, die auf diese Offerte mit einem Angebot antworten können. Der offerierende Agent wählt aus den empfangenen Angeboten das aus seiner Sicht beste Angebot aus und bestätigt dem entsprechenden Agenten die Zusage.

Ein ähnliches Verfahren wird bei Auktionen [10] angewendet, bei denen Agenten für eine Ressource oder eine Aufgabe bieten. Hierbei gibt es verschiedene Formen von Auktionen, bei denen beispielsweise das höchste Gebot oder das erste Gebot den Zuschlag erhält. Auch geheime Gebotsabgaben, bei denen die Bieter die konkurrierenden Angebote nicht kennen, werden eingesetzt.

Wird die Idee der Agentensysteme und der Aushandlung auf die Selbstkonfiguration übertragen, entsprechen die Knoten den Agenten und die Jobs, den bei Agenten üblichen Tasks, die es zu verteilen gilt. Die Übertragung der Verfahrensweise von Agenten auf die Selbstkonfiguration würde bedeuten, dass pro Job, der durch einen Knoten belegt werden soll, mindestens drei Nachrichten notwendig wären. Normalerweise würden jedoch deutlich mehr Nachrichten zustande kommen, da auf eine Offerte mehrere Knoten ein Angebot abgeben würden, aus denen dann das beste Angebot ausgewählt wird.

Diese Vorgehensweise ist sehr aufwändig und die Evaluationen zeigen, dass durch einen einfacheren Mechanismus mehr als die Hälfte der Nachrichten, im Vergleich zum besten Fall des Agentensystems, eingespart werden können. Darüber hinaus gibt es bei der Verhandlung der Agenten das Problem, dass auf eine Offerte ein Angebot mit einer bestimmten Qualität abgegeben wird.

Treffen bei einem Knoten in kurzer Zeit mehrere Offerten ein, auf die er jeweils ein Angebot abgibt, kann es vorkommen, dass ein Angebot noch nicht bestätigt oder abgelehnt wurde, bevor das nächste Angebot abgegeben wird. Damit kann der Agent aber keine zuverlässigen Aussagen über seine Dienstgüte treffen, da er nicht weiß, ob und wie viele Angebote angenommen werden. Lediglich die Dienstgüte des ersten Angebots kann zuverlässig berechnet werden. Jede weitere Dienstgüte hängt davon ab, ob ein vorheriges Angebot angenommen wird oder nicht.

Berechnet der Agent den pessimistischen Fall und geht davon aus, dass das erste Angebot angenommen wird, ist die Dienstgüte des zweiten Angebots unter Umständen zu schlecht, sofern das erste Angebot nicht angenommen wird. Für die Erbringung des Dienstes hätte das grundsätzlich keinen negativen Einfluss, jedoch für die Entscheidung, ob auf das Angebot des Agenten eine Zusage erfolgt oder nicht. Gibt ein Agent fälschlicherweise eine zu geringe Dienstgüte an, bekommt er möglicherweise nicht den Zuschlag. Wird im optimistischen Fall immer die gerade mögliche Dienstgüte im Angebot abgegeben, kann dies dazu führen, dass alle Angebote eines Agenten angenommen werden und damit der Agent nicht mehr in der Lage ist, die Zusagen einzuhalten.

Der Selbstkonfigurationsalgorithmus bietet, besonders für den Fall der Dienstgüte, zuverlässige Zusagen. Die angegebene Dienstgüte einer Dienstbelegung ist zuverlässig, da die Reihenfolge der Dienstbelegungen irrelevant ist. Darüber hinaus sollte die Selbstkonfiguration mit einem einfachen Algorithmus und einer einfachen Metrik umgesetzt werden, um mit möglichst wenig Nachrichten eine bestmögliche Konfiguration zu erstellen. Aus den genannten Gründen hat die vorliegende Implementierung deutliche Vorteile gegenüber dem Einsatz von Agenten.

4.7.2 Distributed Constraint Satisfaction Problem

Eine weitere Möglichkeit, das Problem der Selbstkonfiguration zu lösen, bieten Ansätze des Distributed Constraint Satisfaction Problem (DCSP) [91]. Die DCSPs stellen eine Erweiterung der Aufgabenstellung des Constraint Satisfaction Problem (CSP) [36, 37, 76] auf verteilte Systeme dar.

Beim Constraint Satisfaction Problem geht es darum, eine Belegung von Variablen zu finden, so dass alle Bedingungen (Constraints) erfüllt werden. Im verteilten Fall (DCSP) sind sowohl die Bedingungen, als auch die Variablen auf

verschiedene Knoten oder Agenten verteilt. In [91] wird ein allgemeiner Ansatz zur Lösung der Problemklasse skizziert. Zur Steigerung der Effizienz werden verschiedene Verfahren wie beispielsweise das asynchronous backtracking eingesetzt. Ansatz des Verfahrens ist es, die Belegung der Variablen schrittweise so durchzuführen, dass die erstellte Lösung für möglichst viele Bedingungen erfüllt ist.

Voraussetzung dieser Ansätze ist jedoch, dass eine vollständige Ordnung zwischen den Agenten erstellt wird, um die Weitergabe der Belegungen zu steuern und um Zyklen in der Kommunikation zu vermeiden. Zyklen würden dazu führen, dass eine nicht erfüllbare Bedingung nicht erkannt werden kann. Ist die Ordnung erstellt, kann mit einer relativ geringen Anzahl an Nachrichten eine Belegung ermittelt werden, mit der die Einhaltung der Constraints gewährleistet wird. Der Vorteil der Verfahren liegt darin, dass keine vollständige Vermaschung des Netzwerks gefordert wird.

Problematisch, in Bezug auf die Anzahl der Nachrichten, ist das Verfahren dann, wenn es zu widersprüchlichen Zuweisungen kommt. In diesem Fall wird eine nogood-Nachricht vom Empfängerknoten verschickt, die anzeigt, dass die getroffene Belegung ungültig ist. Außerdem wird keine Aussage über den Aufwand zur Erzeugung der vollständigen Ordnung zwischen den Agenten gemacht.

Die Umsetzung der Selbstkonfiguration durch Abbildung auf ein DCSP würde möglicherweise zu einer besseren Lösung in Bezug auf die Verteilung der Dienste führen, da durch die DCSPs versucht wird, eine optimale Lösung zu erstellen. Der dafür notwendige Aufwand ist jedoch nicht gerechtfertigt. Zum einen muss keine optimale Konfiguration gefunden werden, da die Dienste zur Laufzeit durch die Selbstoptimierung so auf den Knoten verteilt werden, dass die tatsächlich gemessenen Lasten der Dienste optimal verteilt werden, zum anderen ist bereits der initiale Aufwand der Ordnungsbildung in großen Netzen sehr hoch.

Zusätzlich würde die Umsetzung der momentan einfachen Konfigurationsbeschreibung in eine entsprechende Beschreibung mit Bedingungen zu komplexen Konstrukten innerhalb der Konfiguration führen. Der Einsatz von DCSPs würde sich lediglich für die Erweiterung der implementierten Selbstkonfiguration anbieten, wenn zusätzliche Abhängigkeiten zwischen Jobs benötigt würden.

4.7.3 Selbstkonfiguration in Middlewaresystemen

Das Autonomic Networked System [3] ist eine Architektur zur Verwaltung eines ubiquitären Umfelds. Ziel dieser Middleware ist die Kopplung von Geräten anhand von Dienstbeschreibungen. Alle Ressourcen und Geräte werden jeweils durch eine Dienstbeschreibung repräsentiert. Für die Dienstbeschreibung kommt die Ontology Web Language for Services (OWL-S) [9] zum Einsatz.

Am Beispiel eines PDAs, mit dem eine sichere Kommunikationsverbindung aufgebaut werden soll, wird demonstriert, wie die Middleware Dienste miteinan-

der verbindet. Ein soziales Verhalten wie in der vorliegenden Selbstkonfiguration kann durch einen zusätzlichen Dienst implementiert werden. Als Beispiel wird eine Form von Selbstheilung implementiert, bei der ein Dienst so lange nach einem Ersatz für einen ausgefallenen Dienst sucht, bis einer gefunden wurde. Im Fokus der Forschung steht jedoch klar die Kopplung von Diensten und nicht die Implementierung geeigneter Mechanismen zur Realisierung von Selbst-X-Eigenschaften.

Im Gegensatz zur hier vorgestellten Selbstkonfiguration werden im Autonomic Networked System sehr viele Nachrichten erzeugt. Außerdem fehlt eine Beschreibungsmöglichkeit auf Applikationsebene mit der es möglich ist, die Zusammenhänge der Dienste zu beschreiben.

Autonomia [13] ist eine Middleware, die auf den Prinzipien des Autonomic Computing aufbaut. Unter anderem versucht Autonomia einen Ansatz zur Selbstkonfiguration anhand benutzerdefinierter Richtlinien umzusetzen. Anhand der notwendigen Ressourcen und installierter Software kann der Benutzer den Ablauf einer Applikation vorgeben, indem er die Aufgaben modelliert und die Voraussetzungen dafür definiert. Die Informationen über vorhandene Ressourcen und Applikationen können aus dem Applikation Information and Knowledge Repository erfragt werden. Eine modellierte Aufgabe kann als Service Template wiederum in das Repository gespeichert werden.

Anhand der Templates versucht Autonomia, mit Hilfe eines Mobilen Agentensystems basierend auf Jini, die erforderlichen Ressourcen dynamisch zuzuteilen. Die Kommunikation im System wird über einen Event-Server, basierend auf JavaSpaces [18] abgewickelt, an dem sich Agenten für Ereignisse registrieren können.

Der Application Delegation Manager verwaltet und überwacht die Applikationen sowie die Agenten zur Laufzeit. Für die Selbstkonfiguration werden zwei Tabellen benutzt, in denen die Ressourcen der Rechner sowie die Abhängigkeiten der einzelnen Aufgaben hinterlegt sind. Mit diesen Tabellen können die Abhängigkeiten der vom Benutzer definierten Application Service Templates aufgelöst werden. Die Einzelaufgaben können anschließend durch das Mobile Agentensystem verteilt werden.

Der Anwendungsbereich von Autonomia zielt auf die Vereinfachung administrativer Aufgaben in Rechenzentren. Die Selbstkonfiguration innerhalb von Autonomia besitzt mehrere Schwachstellen. Die zentrale Speicherung von Informationen beherbergt die Gefahr eines Single Point of Failure. Der Einsatz von JavaSpaces für die Kommunikation in großen Netzen führt zu Skalierungsproblemen wie in [66] gezeigt wird. Darüber hinaus wird kein Maß für die Güte der Selbstkonfiguration angegeben.

SELFCON [5] beschreibt eine Architektur zur Umsetzung der Selbstkonfigurationseigenschaft auf Netzwerkkomponenten. Unter Verwendung eines LDAP Directory Service wird eine zentrale Instanz mit entsprechenden Informationen aufgebaut, die zur Konfiguration des Netzwerks benutzt wird. In der SELFCON Archi-

tektur sind Netzwerkelemente mit der Fähigkeit zur Selbstkonfiguration vorgesehen, die auf Events reagieren und dadurch eine Selbstkonfiguration des Netzwerks auslösen können. Die Selbstkonfiguration wird anhand von Vorgehensweisen (Policies) gesteuert, die durch die Elemente des Netzwerks umgesetzt werden.

Der Ansatz von SELFCON führt wie bei Autonomia aufgrund der zentralen Informationsverwaltung zu einem Single Point of Failure. In der Architektur wird zwar die Implementierung der Selbstkonfiguration beschrieben, jedoch bleibt die Umsetzung zur Einhaltung der Dienstgüte ungeklärt. Ebenso ist auch bei SELFCON kein Maß für die Bestimmung der Güte der Selbstkonfiguration angegeben.

4.8 Fazit

Die Selbstkonfiguration auf Basis eines kooperativen sozialen Verhaltens zeigt trotz des einfachen Lösungsansatzes sehr gute Ergebnisse. Der vollkommen dezentrale Mechanismus ermöglicht eine einfache Skalierbarkeit auch in Systemen mit einer großen Anzahl an Knoten.

Die Evaluationen in der Middleware bestätigen das erkannte Verhalten der Simulationen und zeigen darüber hinaus noch bessere Werte. Der Unterschied zwischen homogenen und heterogenen Systemen fällt bei den Evaluationen in der Middleware kaum ins Gewicht. Obwohl in bestimmten Konstellationen mehr Nachrichten für die Verteilung der Dienste auf die Knoten benötigt werden, bleibt die durchschnittliche Anzahl an Nachrichten pro Job auf einem sehr niedrigen Wert zwischen 1,1 und 1,6.

Die Simulationen haben gezeigt, dass die Selbstkonfiguration auch für große Netze geeignet ist und die Ergebnisse der Evaluation in der Middleware lassen den Schluss zu, dass das etwas schlechtere Verhalten bei heterogener Hardware in einer realen Umgebung vermutlich nicht zum Tragen kommt.

Ansätze basierend auf Agentensystemen benötigen für eine Selbstkonfiguration nach dem vorgestellten Muster weitaus mehr Nachrichten als die Selbstkonfiguration durch kooperatives soziales Verhalten, da sie für jeden Job mindestens drei Nachrichten benötigen, im Normalfall jedoch deutlich mehr Nachrichten erzeugen.

Die Umsetzung der Selbstkonfiguration ist bewusst sehr einfach gehalten, um zu zeigen, dass das Verfahren bereits mit minimalem Aufwand zu guten Ergebnissen führt. Die vorgestellte Form der Selbstkonfiguration beinhaltet noch keine Optimierungen, die zusätzliche Sicherheit bei der Verteilung der Jobs gewährleisten oder zu weniger Nachrichten führen. Es gibt hier eine Reihe von Optimierungen, die eingesetzt werden könnten, um beispielsweise in der Verifikationsphase einen Großteil der aufgetretenen Nachrichten zu eliminieren. Es könnten mit jeder Belegungsnachricht die letzten empfangenen Belegungen nochmals verschickt

werden, um Redundanz bei den Informationen und somit zusätzliche Sicherheit gegenüber Nachrichtenverlust zu gewährleisten.

5 Selbstoptimierung

Bei der Betrachtung der Kommunikationsmechanismen des menschlichen Körpers fällt auf, dass unterschiedliche Systeme für die Übertragung von Informationen benutzt werden. Unter anderem unterscheiden sich die Systeme in der Geschwindigkeit, mit der Information verbreitet werden kann und der Menge der zu reizenden Zellen. Für spontane und sehr schnelle Reaktionen wird das somatische und das vegetative Nervensystem benutzt, die beide gezielt Bereiche des Körpers, wie beispielsweise Muskulatur, aktivieren können. Das Hormonsystem hingegen wirkt langsamer und gegebenenfalls auf größere Bereiche des Körpers.

Der Vergleich zwischen dem menschlichen Körper und modernen Softwaresystemen liegt sehr nahe. Die schnelle und gezielte Reaktion auf ein Ereignis wird, wie beim menschlichen Körper, durch eine gezielte Nachricht, oder einen Methodenaufruf einer Komponente erreicht. Das somatische und vegetative Nervensystem kann also mit den aktiven Kommunikationsmechanismen verglichen werden. Da aktuelle Systeme über keine Selbstorganisationsmechanismen verfügen, fehlt ein vergleichbarer Ansatz zum menschlichen Hormonsystem. In diesem Kapitel wird der Ansatz zu einem derartigen System erläutert und evaluiert. Das künstliche Hormonsystem soll dabei, wie das natürliche Vorbild auch, vollständig ohne globale Kontrollinstanz auskommen und nur mit lokalem Wissen arbeiten, um das Selbstoptimierungsziel zu erreichen.

Nachfolgend wird ein Überblick über das menschliche Hormonsystem gegeben, soweit es für die Modellierung des künstlichen Hormonsystems von Bedeutung ist. Anschließend wird das Modell des künstlichen Hormonsystems [75] erläutert und anhand einer Lastoptimierung werden verschiedene Metriken evaluiert. Danach wird die Integration in die Middleware beschrieben und evaluiert.

Die nachfolgend beschriebene Simulationsumgebung, die Einfach- und gewichtete Übergabestrategie sowie die Übergabestrategie mit adaptiver Schranke wurden im Rahmen der Diplomarbeit „Selbstorganisation der Knoten einer Peer-to-peer-Middleware mittels Botenstoffen“ [67] implementiert.

5.1 Das menschliche Hormonsystem

Das Hormonsystem wird, ebenso wie das somatische und das vegetative Nervensystem, zur Verbreitung von Informationen beziehungsweise von Reizen im

Körper benutzt. Im Gegensatz zum somatischen Nervensystem, das überwiegend bewusst gesteuert wird, sind das vegetative Nervensystem und das Hormonsystem weitestgehend der bewussten Kontrolle entzogen. Die drei Systeme unterscheiden sich weiterhin in der Geschwindigkeit, mit der Informationen verbreitet werden. Während das somatische und das vegetative Nervensystem Reize über Nervenbahnen leiten, benutzt das Hormonsystem chemische Botenstoffe, die in den Blutkreislauf oder das umgebende Gewebe abgegeben werden. Verglichen mit der Reizleitung auf Nervenbahnen, ist die Verbreitung der Hormone durch den Blutkreislauf relativ langsam. Allerdings können durch das Hormonsystem sehr große Bereiche des Körpers gezielt aktiviert werden.

Neben den klassischen Hormonen, die von einer spezialisierten Drüse, beispielsweise der Bauchspeicheldrüse, gebildet und in den Blutkreislauf abgegeben werden, um ihre Wirkung an anderen Stellen des Körpers zu entfalten, werden noch eine Reihe anderer Botenstoffe unterschieden. Gewebshormone, wie beispielsweise die Hormone des Verdauungstraktes (gastrointestinale Hormone), wirken lokal in dem Gewebe, in dem sie produziert werden. Als Mediatoren werden Botenstoffe bezeichnet, die nicht in den Blutkreislauf ausgeschüttet werden, sondern direkt auf benachbarte Zellen (parakrin) wirken. Beeinflusst der sezernierte Mediator auch die produzierende Zelle selbst, spricht man von autokriner Wirkung. Interleukine stellen einen Spezialfall der Mediatoren dar, sie werden zwischen den Leukozyten der Immunabwehr ausgetauscht. Eine letzte Gruppe bilden die Neurotransmitter. Sie sind Hilfsstoffe des Nervensystems, die den nervösen Impuls auf chemischem Weg durch den synaptischen Spalt von einer Nervenzelle zur nächsten weiterleiten.

Obwohl die meisten Hormone über den Blutkreislauf theoretisch jede Zelle des Körpers erreichen können, entfalten sie ihre Wirkung nur in den Zellen, in denen die zu dem Hormon korrespondierenden Rezeptoren vorhanden sind. Die Hormone binden an die Rezeptoren und leiten so die Information auf chemischem Weg in das Zellinnere weiter. Die Rezeptorstrategie ermöglicht es, einerseits die Wirkung der Hormone auf die gewünschten Zellen einzuschränken und andererseits durch dasselbe Hormon in unterschiedlichen Zellen unterschiedliche Reaktionen hervorzurufen.

Die meisten hormonellen Regelkreise besitzen mindestens zwei Hormone, die an der Erzeugung einer Reaktion beteiligt sind. Ein aktivierendes Hormon veranlasst Zellen zu einer Reaktion, während ein unterdrückendes Hormon die Reaktion dämpft oder gegebenenfalls beendet. Die Reaktion der betroffenen Zellen hängt von der zeitlichen Verteilung der Hormonkonzentrationen ab. Befinden sich die aktivierenden und unterdrückenden Hormone im Gleichgewicht, erzeugen die betroffenen Zellen keine oder eine normale Reaktion. Wird ein Überschuss der aktivierenden Hormone erzeugt, wird die Reaktion der Zellen verstärkt. Meistens wird dadurch auch die Produktion der unterdrückenden Hormone angeregt, die zu einer Normalisierung beziehungsweise zur Beendigung der Reaktion führt.

Weitergehende Betrachtungen der Funktionsweise des humanen Nerven- und Hormonsystems sind in der medizinischen Fachliteratur wie dem Taschenatlas der Physiologie [60], dem Buch Neuroanatomie Struktur und Funktion [68] sowie im Buch zur Biochemie des Menschen [23] zu finden.

5.2 Modell des künstlichen Hormonsystems

5.2.1 Vereinfachungen zur Modellbildung

Auf der Suche nach einfachen Mechanismen zur Realisierung von Selbstorganisationsmechanismen scheint eine direkte Übertragung der Mechanismen des menschlichen Hormonsystems nicht sinnvoll. Die Berechnung der Konzentrationsunterschiede ist komplex und mit einem hohen Rechenaufwand verbunden. Aus diesem Grund muss das Verfahren so weit vereinfacht werden, dass es mit möglichst geringem Aufwand zufrieden stellende Ergebnisse liefert. Die grundlegende Arbeitsweise des Hormonsystems, Information auf ein bereits vorhandenes System (den Blutkreislauf) aufzuprägen, kann in praktisch allen Kommunikationssystemen realisiert werden, da bereits Nachrichten zwischen den Knoten oder Komponenten der Systeme ausgetauscht werden.

Die Umsetzung der Hormonkonzentrationen und die Nutzung verschiedener Hormone kann durch Verwendung einzelner Parameter und deren Normierung erzielt werden. Die Übertragung der Werte zu unterschiedlichen Knoten des Netzwerks wird dadurch realisiert, dass die lokalen Informationen auf die Nachrichten, die zwischen den Knoten ausgetauscht werden, aufgeprägt und beim Empfänger ausgewertet werden. In der Middleware können diese Informationen durch Monitore auf ausgehende Nachrichten aufgeprägt und beim Empfänger durch Monitore für eingehende Nachrichten empfangen werden.

Verglichen mit dem menschlichen Hormonsystem bedeutet dies, dass die Monitore für ausgehende Nachrichten den hormonproduzierenden Drüsen entsprechen, die normierten Parameter den Hormonen, und die Monitore für eingehende Nachrichten den Rezeptoren der Zellen. Die Verarbeitung der Information erfolgt lokal auf den Knoten, womit die Knoten mit den Zellen verglichen werden können.

5.2.2 Grundlagen der Modellbildung

Bei der Modellierung des künstlichen Hormonsystems wird bewusst ein allgemeiner Ansatz gewählt, um den Mechanismus auf möglichst viele Anwendungsbereiche abbilden zu können. Aus diesem Grund wird zunächst von den Gegebenheiten der Middleware abstrahiert und der mathematische Ansatz allgemein dargestellt. Allerdings bestimmt die Middleware die grundlegende Struktur des Systems mit

Knoten und Diensten. Für die Evaluierung des allgemeinen Ansatzes wird ein Simulator benutzt, mit dem verschiedene Metriken getestet und bewertet werden können. Die Integration des Mechanismus in die Middleware zur Evaluierung wird in Kapitel 5.6 beschrieben.

Das Modell des künstlichen Hormonsystems beschreibt die Abbildung von Systemparametern und deren Verarbeitung durch Metriken zur Erzeugung einer Reaktion der Knoten. Die Metriken bestimmen somit die Reaktion der Knoten in Abhängigkeit der empfangenen und der eigenen Systemparameter. Ab Kapitel 5.3 werden verschiedene Metriken zur Lastoptimierung eines Systems dargestellt.

Gegeben sei ein System, bestehend aus n Knoten, auf denen Dienste s ausgeführt werden können. Ein Knoten besitze die Möglichkeit m Systemparameter zu messen, die jeweils durch eine Ressource r_i dargestellt werden. Für jede Ressource gilt $0 \leq r_\mathrm{i} \leq 100$ mit $1 \leq i \leq m$. Damit werden die Ressourcen auf einen Zahlenbereich von 0 bis 100 eingeschränkt. Die tatsächlich gemessenen Werte der Systemparameter müssen für jede Ressource r_i auf diesen Bereich normiert werden.

Der Anteil der Ressource r_i der durch einen Dienst s belegt wird ist $r_\mathrm{i}(s)$. Der gesamte Ressourcenverbrauch einer Ressource r_i auf einem Knoten k berechnet sich aus der Summe der Anteile aller Dienste auf diesem Knoten. Sei $S(k)$ die Menge der Dienste s auf dem Knoten k, dann gilt

$$R_\mathrm{i}(k) = \sum_{s \in \mathcal{S}(k)} r_i(s) \tag{5.1}$$

$$0 \leq R_\mathrm{i}(k) \leq 100 \tag{5.2}$$

mit $1 \leq k \leq n$ und $1 \leq i \leq m$. Formel 5.2 stellt sicher, dass die Belegung einer Ressource den Wert 100 nicht übersteigt und somit für jede Ressource und jeden Knoten der Wertebereich aller Ressourcen auf den Normierungsbereich beschränkt ist.

5.3 Metriken

Metriken berechnen anhand der empfangenen Parameter die Reaktion des Knotens, vergleichbar der Reaktion der Zellen auf die vorhandene Hormonkonzentration. Da die zu erzeugende Reaktion von der Anwendung abhängt, müssen die Metriken jeweils für den entsprechenden Anwendungsfall implementiert werden.

Die nachfolgend beschriebenen Metriken ermöglichen eine Lastoptimierung zwischen den Knoten. Dabei sollen die Dienste so auf die vorhandenen Knoten verteilt werden, dass eine möglichst gleichmäßige Last auf allen Knoten entsteht. Wichtig ist, dass es für diese Art der Lastverteilung keine zentrale Kontrolle gibt.

Für das Verständnis der Metriken ist die Verteilung der Information entscheidend. Es wird davon ausgegangen, dass jeder Knoten seine aktuelle Last durch die Metrik berechnet und an ausgehende Nachrichten anhängt. Durch diesen gerichteten Nachrichtenversand können pro Nachricht zwei Knoten identifiziert werden. Der Knoten k_{send}, der seine Information an eine Nachricht anhängt und der Knoten Knoten k_{receive}, an den die Nachricht gesendet wird. Somit ist k_{receive} in der Lage, mit den erhaltenen Informationen anhand der Metriken eine entsprechende Reaktion zu berechnen.

Die nachfolgend beschriebenen Metriken werden als Übergabestrategien bezeichnet, weil es sich dabei um Verfahren handelt, mit denen entschieden wird, ob zwischen zwei Knoten ein Dienst und damit Last übergeben werden soll. Dabei ist zu beachten, dass eine Nachricht zwar vom Knoten k_{send} an den Knoten k_{receive} gesendet wird, aber ein Dienst gegebenenfalls in entgegengesetzter Richtung von k_{receive} zu k_{send} verlegt wird, da der Knoten k_{receive} die Reaktion berechnet.

5.3.1 Einfache und gewichtete Übergabestrategie

Die einfache und die gewichtete Übergabestrategie verfolgen den Ansatz, eine unterschiedliche Last zwischen den Knoten auszugleichen, unabhängig von der Höhe des zu erzielenden Gewinns. Die Last eines Knotens $nload_{\text{avg}}(k)$ berechnet sich aus dem Durchschnitt der Summe der einzelnen Ressourcen. Jede Ressource besitzt ein ganzzahliges Gewicht $w_{\text{i}} \geq 1$ wobei $1 \leq i \leq m$ mit dem die Ressource in der Berechnung der Gesamtlast gewichtet werden kann. Die einfache Übergabestrategie ist ein Spezialfall der gewichteten Übergabestrategie, bei der alle Gewichte $w_{\text{i}} = 1$ sind.

$$nload_{\text{avg}}(k) = \frac{\sum_{i=1}^{k} w_i \cdot R_i(k)}{\sum_{i=1}^{k} w_i} \tag{5.3}$$

Für den Sendeknoten k_{send} und den Empfängerknoten k_{receive} ergeben sich damit die Lasten $nload_{\text{avg}}(k_{\text{send}})$ beziehungsweise $nload_{\text{avg}}(k_{\text{receive}})$. Die Differenz der beiden Lasten ergibt die Lastdifferenz zwischen den beiden Knoten.

$$d_{\text{load}} = nload_{\text{avg}}(k_{\text{receive}}) - nload_{\text{avg}}(k_{\text{send}}) \tag{5.4}$$

Ist die Lastdifferenz d_{load} negativ, ist die Last des Sendeknotens größer als die Last des Empfängerknotens. In diesem Fall wird kein Dienst verlegt, da dies zu einer weiteren Verschlechterung der Lastverhältnisse führen würde. Um das vorhandene Lastverhältnis zu minimieren, muss der Knoten k_{receive} einen Dienst

finden, dessen Last im besten Fall genau $d_{\text{load}}/2$ ist. In diesem Fall würde die Last zwischen den beiden Knoten genau ausgeglichen. Die Last, die ein Dienst erzeugt, wird äquivalent zur Knotenlast wie folgt berechnet:

$$sload_{\text{avg}}(s) = \frac{\sum_{i=1}^{k} w_i \cdot r_i(k)}{\sum_{i=1}^{k} w_i} \tag{5.5}$$

Somit kann für jeden Dienst $s \in S$ auf einem Knoten die Distanz $\Delta(s)$ zum optimalen Wert für den gesuchten Lastausgleich berechnet werden.

$$\Delta(s) = \left| \frac{d_{\text{load}}}{2} - sload_{\text{avg}}(s) \right| \tag{5.6}$$

Für die Verlegung wird der Dienst benutzt, für den $\Delta(s)$ minimal ist und

$$\Delta(s) < \frac{d_{\text{load}}}{2} \tag{5.7}$$

gilt. Für den Fall, dass Ungleichung 5.7 nicht erfüllt ist, wird zwar immer noch ein Lastausgleich erzielt solange $\Delta(s) < d_{\text{load}}$ gilt, jedoch wird der Knoten k_{send} dadurch über das mittlere Maß hinaus belastet.

5.3.2 Evaluierung

Evaluationsmethodik

Zur Evaluation der Übergabestrategien wurde ein Simulator in Java implementiert, der das Verhalten der Middleware nachbildet und darauf aufbauend den Selbstorganisationsmechanismus umsetzt. Der Mechanismus wird benutzt, um die Last in einem Netzwerk anhand der drei Ressourcen CPU-Auslastung *cpu*, Speicherverbrauch *mem* und Kommunikationsbandbreitenverbrauch *com* zu balancieren.

Der Simulator kann eine gewünschte Anzahl an Knoten und Diensten generieren, zwischen denen eine Kommunikation simuliert wird, auf deren Nachrichten die Lastinformationen aufgeprägt werden. Abbildung 5.1 zeigt den Simulator nach der Initialisierung von 1000 Knoten.

Jeder Knoten wird durch ein Kästchen mit einem dreifarbigen Kreis dargestellt. Jede Farbe repräsentiert eine der Ressourcen, wobei die Intensität der Farbe die Menge der belegten Ressource widerspiegelt. Je intensiver eine Farbe ist,

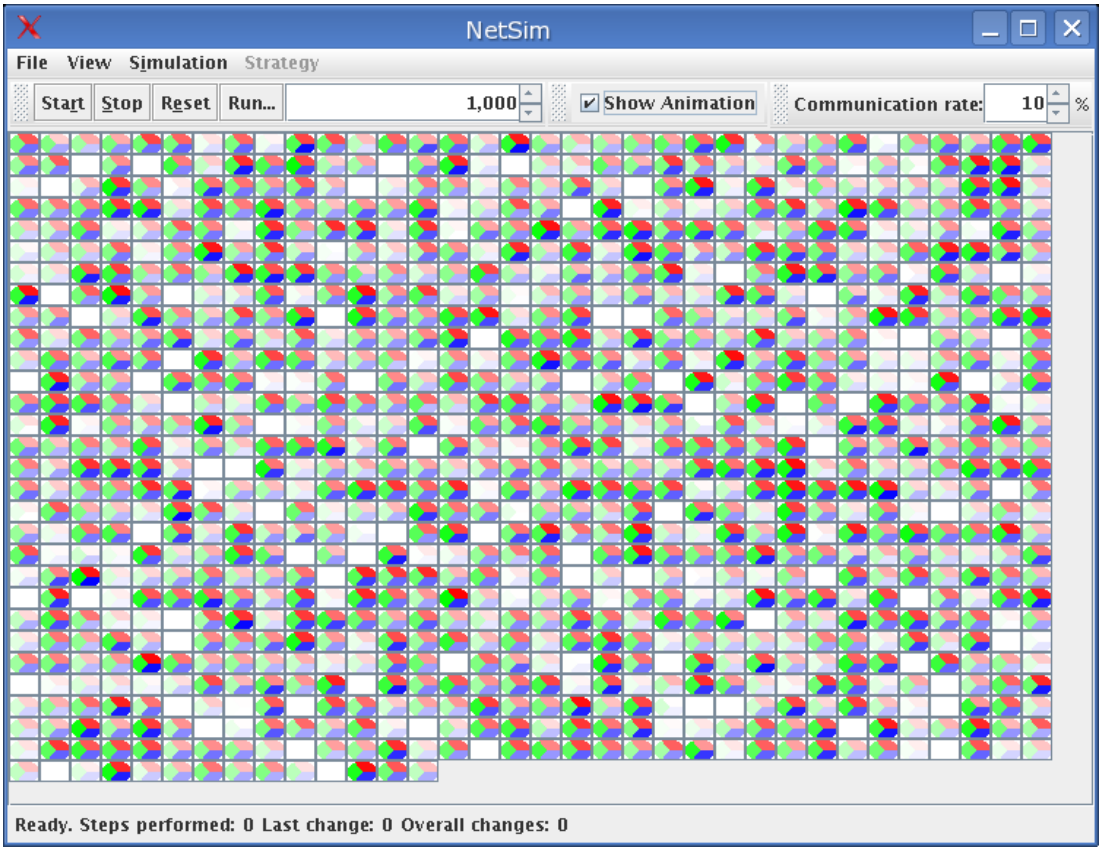

Abbildung 5.1: Simulator nach der Initialisierung von 1000 Knoten

desto mehr ist von einer Ressource verbraucht. Der Simulator ermöglicht die Auswertung sowohl der einzelnen Knoten wie auch des gesamten Netzes. Dadurch können die erzielten Ergebnisse dem rechnerischen Optimum gegenübergestellt und mit diesem verglichen werden.

Initialisierung der Simulation

Eine Simulation wird initialisiert, indem die Anzahl der Knoten, die maximalen Lastparameter und die Optimierungsstrategie gewählt werden. Damit erstellt der Simulator zunächst die Anzahl an Knoten und initialisiert jeden Knoten mit einer Anfangslast, die sich aus den darauf befindlichen Diensten errechnet.

Für die Knoten kann die Maximalkapazität aller drei Ressourcen durch $load_{\max}$ definiert werden. Für jeden Knoten wird damit zufällig eine initiale Maximalkapazität zwischen 0 und $load_{\max}$ erzeugt.

Algorithmus 3 Initialisierung des Simulators

```
1: for k = 1 to n do
2:   while capacity of node k is not expired do
3:     r_cpu(s_new) = random(0, cpu_max)
4:     r_mem(s_new) = random(0, mem_max)
5:     r_com(s_new) = random(0, com_max)
6:     if (R_cpu(k) + r_cpu(s_new) ≤ load_max) and
          (R_mem(k) + r_mem(s_new) ≤ load_max) and
          (R_com(k) + r_com(s_new) ≤ load_max) then
7:       add service s_new to node k
8:     else
9:       capacity of node k expired
10:    end if
11:  end while
12: end for
```

Bei der Erzeugung der Dienste werden die drei Parameter $cpu_{\max}$, $mem_{\max}$ und $com_{\max}$ definiert, die den maximalen Verbrauch jeder Ressource pro Dienst angeben. Der tatsächliche Verbrauch der drei Ressourcen wird jeweils zufällig zwischen 0 und dem jeweiligen Maximalwert der Ressourcen gewählt.

Algorithmus 3 zeigt den Ablauf der Initialisierung aller Knoten. Für jeden Knoten wird geprüft, ob er noch freie Ressourcen besitzt (Zeile 2). Dann wird ein neuer Dienst mit zufälligem Ressourcenverbrauch erzeugt (Zeilen 3 bis 5). Anschließend wird geprüft, ob der Knoten noch ausreichend Ressourcen für den Dienst besitzt (Zeile 6). Wenn dies der Fall ist, wird der Dienst dem Knoten zugewiesen (Zeile 7), andernfalls ist die Belegung des Knotens abgeschlossen und der nächste Knoten wird initialisiert.

Modellierung der Kommunikation

Das verwendete Kommunikationsmodell stützt sich auf die Annahme, dass bestimmte Dienste aufgrund der Applikationsstruktur bevorzugt miteinander kommunizieren, während es zwischen anderen Diensten nicht oder nur selten zum Austausch von Nachrichten kommt. Dieses Verhalten ist aus der Struktur der Klassenabhängigkeiten objektorientierter Software abgeleitet. Gleiches gilt für verteilte Anwendungen, basierend auf Remote Procedure Call (RPC) beziehungsweise auf Remote Method Invocation (RMI), bei denen die Aufrufstruktur weitestgehend fest vorgegeben ist. Resultierend aus diesen Voraussetzungen wird jedem Dienst zu Beginn der Simulation eine feste Menge an Kommunikationspartnern zugeordnet, deren Größe im Bereich von 1 bis $cp_{\max}$ mit wählbarer oberer Schranke $cp_{\max}$ liegt.

Tabelle 5.1: Standardparameter der Simulationen

Parameter	Wert	Beschreibung
n	1000	Anzahl Knoten
c	0,1	Prozentualer Anteil an Nachrichten in Bezug auf n
$com_{\max}$	10	Maximale Kommunikationsbandbreite $b(s)$
$cpu_{\max}$	10	Maximale CPU-Last $l(s)$
$mem_{\max}$	10	Maximaler Speicherverbrauch $m(s)$
$tc_{\max}$	0	Maximale Transferkosten $tc(s)$
$cp_{\max}$	50	Maximale Anzahl Kommunikationspartner
$load_{\max}$	100	Maximale Knotenkapazität

In jedem Schritt der Simulation wird eine festgelegte Anzahl von Diensten als Ausgangspunkt für die Kommunikation ausgewählt. Zu jedem dieser Dienste wird dann aus dessen Kommunikationspartnern ein Empfängerdienst bestimmt. Dies führt pro Simulationsschritt zu n verschiedenen gerichteten Kommunikationen zwischen jeweils zwei Knoten. Andersherum betrachtet repräsentiert ein Simulationsschritt die Zeitspanne, während der im realen Netz n Kommunikationsvorgänge beobachtet werden können.

Es wird davon ausgegangen, dass im realen Netzwerk die Möglichkeit von Broadcast-Nachrichten besteht. Knoten, die keine Dienste beherbergen, nutzen diese Möglichkeit, um in bestimmten Abständen Nachrichten an andere Knoten des Netzes zu senden. Andernfalls wären diese Knoten vom Optimierungsprozess ausgenommen, obwohl gerade sie über ein Höchstmaß an freien Kapazitäten verfügen. In der Simulation werden hierzu pro Schritt mit einer bestimmten Wahrscheinlichkeit leere Knoten als Sender ausgewählt. Da die Menge der Zielknoten in diesem Fall nicht durch die vorhandenen Dienste bestimmt ist, wird ein zufälliger Knoten als Ziel der Nachricht ausgewählt.

Simulationsparameter und Auswertung

Jede Simulation wurde mit den gleichen Einstellungen 100-mal durchgeführt. Auf diese Weise wird zum einen gezeigt, dass die Mechanismen auch bei mehrfacher Anwendung ein stabiles Verhalten zeigen und zum anderen, dass die Güte nicht von einer gewählten Anfangskonfiguration abhängt, da jede Simulation mit neuen Werten im gegebenen Rahmen zufällig initialisiert wird. Sofern nicht ausdrücklich andere Werte angegeben werden, wurden für die Simulationen die in Tabelle 5.1 beschriebenen Werte benutzt.

Für die Berechnung der Simulationsergebnisse wird das arithmetische Mittel benötigt, das mit einem Strich über dem Wert dargestellt wird (z.B. $\overline{x}$), die absolute Abweichung mit δ_x und die Standardabweichung mit σ_x.

Zur Beurteilung der Güte des Selbstorganisationsmechanismus wird das arithmetische Mittel aller Knotenlasten $\overline{nload_{\text{avg}}}$ (der Erwartungswert) berechnet, um damit den Mittelwert der absoluten Abweichungen aller Knoten $\delta_{nload_{\text{avg}}}$ über alle Simulationsläufe ermitteln zu können. Die absolute Abweichung gibt an, wie weit der tatsächlich gemessene Wert von dem Erwartungswert im Durchschnitt abweicht.

$$\overline{nload_{\text{avg}}} = \frac{1}{n} \sum_{i=1}^{n} nload_{\text{avg}}(i) \tag{5.8}$$

$$\delta_{nload_{\text{avg}}} = \frac{1}{n} \sum_{i=1}^{n} \left| nload_{\text{avg}}(i) - \overline{nload_{\text{avg}}} \right| \tag{5.9}$$

Da die Simulation jeweils mehrfach durchgeführt wird, wird das arithmetische Mittel der absoluten Abweichungen berechnet. Für x Simulationsläufe lautet die Formel wie folgt:

$$\overline{\delta_{nload_{\text{avg}}}} = \frac{1}{x} \sum_{i=1}^{x} \delta_{nload_{\text{avg}}(i)} \tag{5.10}$$

Zusätzlich wird als Maß für die Streuung der Messwerte die Standardabweichung der 100 gemessenen Simulationen berechnet. Wenn die Standardabweichung gering ist bedeutet dies, dass sich alle Messwerte in einem engen Bereich um den Erwartungswert scharen.

$$\sigma_{nload_{\text{avg}}} = \sqrt{\frac{1}{x} \sum_{i=1}^{x} (\delta_{nload_{\text{avg}}(i)} - \overline{\delta_{nload_{\text{avg}}}})^2} \tag{5.11}$$

Einfache Übergabestrategie

In Tabelle 5.2 sind die Ergebnisse von 100 Simulationen abgebildet. Die erste Spalte gibt die Anzahl der Simulationsschritte an, bei denen die übrigen Werte ermittelt wurden. Die zweite Spalte $\bar{t}$ gibt die durchschnittliche Anzahl verlegter Dienste an und die dritte Spalte die Standardabweichung der Verlegungen.

Die vierte und fünfte Spalte geben die mittlere absolute Abweichung der Knotenlast und die zugehörige Standardabweichung an. Die mittlere Knotenlast $\overline{nload_{\text{avg}}}$

Tabelle 5.2: Einfache Übergabestrategie

Schritte	$\bar{t}$	σ_t	$\overline{\delta_{nload_{\text{avg}}}}$	$\sigma_{\delta_{nload_{\text{avg}}}}$
0	0,00	0,00	22,95	0,43
100	4398,58	58,09	1,73	0,09
200	5131,64	90,90	0,78	0,02
300	5243,87	93,32	0,70	0,02
400	5278,10	94,44	0,68	0,02
500	5292,04	94,95	0,67	0,02
1000	5309,42	95,27	0,66	0,02
2000	5313,65	95,35	0,66	0,02

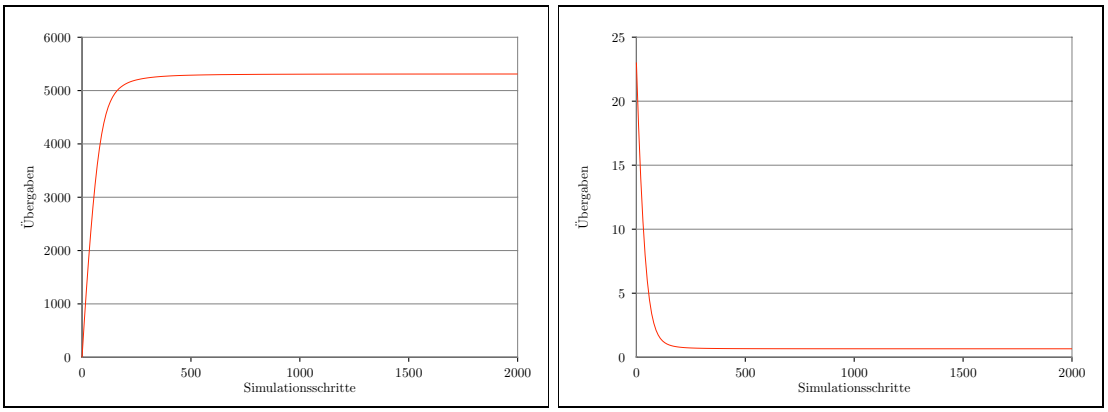

Abbildung 5.2: Anzahl der Dienstverlegungen $\bar{t}$ (links) und mittlerer Fehler $\overline{\delta_{nload_{\text{avg}}}}$ (rechts) bei der einfachen Übergabestrategie

liegt bei 41,15. Daraus ergibt sich, dass nach 2000 Simulationsschritten das rechnerische Optimum zu 98,4% erreicht wurde. Es gilt zu beachten, dass der rechnerische Optimalwert nicht erreicht werden kann, da jeder Knoten aufgrund der zufälligen Initialisierung der Ressourcen pro Dienst, unterschiedliche Lasten für die einzelnen Ressourcen besitzt. Die Standardabweichung $\sigma_{\delta_{nload_{\text{avg}}}}$ von 0,02 zeigt jedoch, dass alle Messwerte sehr nahe bei dem Erwartungswert liegen woraus abgeleitet werden kann, dass die Optimierungsstrategie nicht nur sehr gut den Erwartungswert erreicht, sondern dieser auch stabil erreicht wird.

Aus der Tabelle ist zu erkennen, dass die Optimierung bereits nach 200 Simulationsschritten 96,5% der Dienstverlegungen durchgeführt hat und nach 500 Schritten mit 99,6% praktisch abgeschlossen ist. Abbildung 5.2 zeigt das Verhalten der Dienstverlegungen und der Standardabweichung in zwei Diagrammen. Es ist deutlich zu erkennen, dass die Optimierungsstrategie sehr rasch konvergiert.

Tabelle 5.3: Werte der drei Ressourcen bei der einfachen Übergabestrategie

Schritte	$\bar{t}$	$\overline{\delta_{com}}$	$\sigma_{\delta_{com}}$	$\overline{\delta_{cpu}}$	$\sigma_{\delta_{cpu}}$	$\overline{\delta_{mem}}$	$\sigma_{\delta_{mem}}$
0	0,00	23,47	0,47	23,43	0,44	23,46	0,46
100	4398,58	5,66	0,16	5,67	0,13	5,66	0,14
200	5131,64	5,42	0,14	5,41	0,13	5,40	0,14
300	5243,87	5,41	0,14	5,39	0,13	5,38	0,14
400	5278,10	5,40	0,14	5,39	0,13	5,38	0,13
500	5292,04	5,40	0,14	5,39	0,13	5,38	0,14
1000	5309,42	5,40	0,15	5,39	0,13	5,38	0,14
2000	5313,65	5,40	0,14	5,39	0,13	5,38	0,14

In Tabelle 5.3 sind zu den Werten aus Tabelle 5.2 die Werte der einzelnen Ressourcen r_{com}, r_{cpu} und r_{mem} dargestellt. Die Tabelle zeigt ebenso wie Abbildung 5.2 ein sehr schnelles Konvergenzverhalten der einzelnen Ressourcen.

Gewichtete Übergabestrategie

Die einfache Übergabestrategie stellt, wie bereits erwähnt, einen Spezialfall der gewichteten Übergabestrategie dar. Um anwendungsspezifische Anforderungen abbilden zu können, werden mit der gewichteten Übergabestrategie die einzelnen Ressourcen unterschiedlich stark bewertet.

Tabelle 5.4 zeigt die Simulationsergebnisse für unterschiedliche Gewichte w_{com} der Kommunikationsbandbreite r_{com}. Die Gewichte für w_{cpu} und w_{mem} sind bei allen Simulationen 1. Die erste Zeile mit $w_{\text{com}} = 1$ entspricht dem Fall der einfachen Übergabestrategie.

Die Simulationsergebnisse zeigen, dass die Optimierungsstrategie den stärker gewichteten Parameter entsprechend gut optimiert, wohingegen die beiden anderen Parameter in etwa dem gleichen Maß schlechter optimiert werden. Dass sich eine Verbesserung bei der Gesamtlast $\overline{\delta_{nload_{\text{avg}}}}$ ergibt liegt daran, dass der Parameter w_{com} im Verhältnis eine größere Verbesserung erfährt. Der mittlere Fehler der Ressource r_{com} wird bei einem Gewicht von 5 auf nahezu die Hälfte reduziert, wohingegen die mittleren Fehler der beiden Ressourcen r_{cpu} und r_{mem} jedoch nicht um das Doppelte wachsen. Daraus ergibt sich insgesamt eine Reduzierung des mittleren Fehlers. Insgesamt werden jedoch für die besseren Ergebnisse auch mehr Dienstverlegungen durchgeführt, wie in der Spalte $\bar{t}$ zu erkennen ist.

Die Standardabweichung $\sigma_{\delta_{nload_{\text{avg}}}}$ bleibt bei allen Simulationen auf dem gleich guten Wert von 0,02. Für die einzelnen Parameter ergeben sich zwar entsprechende Verbesserungen beziehungsweise Verschlechterungen, die sich in der Gesamtbetrachtung jedoch ausgleichen.

Tabelle 5.4: Gewichtete Übergabestrategie mit verschiedenen Gewichten für w_{com}

w_{com}	$\bar{t}$	$\overline{\delta_{nload_{\text{avg}}}}$	$\sigma_{\delta_{nload_{\text{avg}}}}$	$\overline{\delta_{com}}$	$\sigma_{\delta_{com}}$	$\overline{\delta_{cpu}}$	$\sigma_{\delta_{cpu}}$	$\overline{\delta_{mem}}$	$\sigma_{\delta_{mem}}$
1	5313,65	0,66	0,02	5,40	0,14	5,39	0,13	5,38	0,14
2	5430,39	0,63	0,02	4,42	0,12	6,32	0,15	6,30	0,16
3	5578,49	0,60	0,02	3,85	0,09	7,18	0,18	7,15	0,21
4	5718,78	0,58	0,02	3,35	0,08	7,77	0,19	7,79	0,19
5	5827,94	0,57	0,02	2,95	0,08	8,21	0,23	8,22	0,22

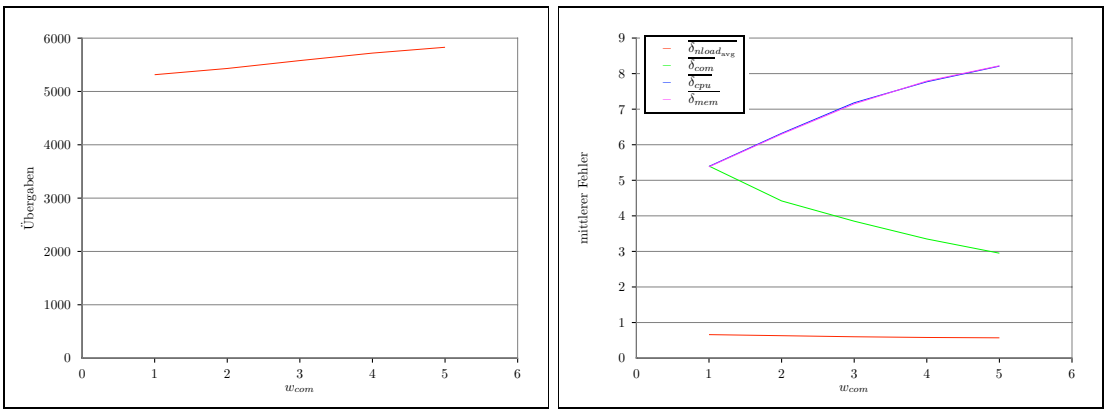

Abbildung 5.3: Anzahl der Dienstverlegungen $\bar{t}$ (links) und mittlerer Fehler $\overline{\delta_{nload_{\text{avg}}}}$ (rechts) bei unterschiedlichen Gewichten für w_{com}

In Abbildung 5.3 ist das eben erläuterte Verhalten der Anzahl der Dienstverlegungen, des mittleren Fehlers insgesamt sowie des mittleren Fehlers der einzelnen Parameter in Abhängigkeit der unterschiedlichen Gewichte für w_{com} nochmals in zwei Diagrammen visualisiert.

Zusammenfassend lässt sich anhand der Simulationsergebnisse feststellen, dass der entwickelte Selbstorganisationsmechanismus die gestellte Aufgabe der Lastoptimierung sehr gut erfüllt. Dabei kommt der Algorithmus ohne jegliche zentrale Kontrolle aus und arbeitet ausschließlich auf den lokal gesammelten Daten. Außerdem wird für die Übertragung der Daten der bereits vorhandene Kommunikationsmechanismus benutzt, auf den lediglich zusätzliche Information aufgeprägt wird.

5.3.3 Übergabestrategie mit adaptiver Schranke

Wie erwartet und durch die Evaluierungen belegt, zeigen die einfache und die gewichtete Übergabestrategie für den Lastausgleich ein sehr gutes Konvergenzverhalten hin zum rechnerischen Optimalwert. Da jedoch immer ein Dienst verlegt wird, sobald ein Lastausgleich möglich ist, werden sehr viele Dienste transferiert.

Manche Dienste werden sogar mehrfach verlegt bis ein Gleichgewicht im Netzwerk entsteht.

Ausgehend von den ermittelten Durchschnittswerten der einfachen Übergabestrategie kann eine Abschätzung berechnet werden, wie viele Dienste verlegt werden müssen, um das rechnerische Optimum zu erreichen. Die mittlere Last eines Knotens beträgt bei den Simulationen etwa 41,15. Eine Gleichverteilung bei den Zufallswerten vorausgesetzt bedeutet, dass jeder Dienst eine mittlere Last von fünf erzeugt. Damit ist jeder Knoten mit acht Diensten belegt.

Die mittlere absolute Abweichung von 22,95 zu Beginn der Simulation besagt, dass im Mittel vier Dienste zu viel, beziehungsweise zu wenig auf den Knoten vorhanden sind. Manche Knoten sind also mit zwölf, andere nur mit vier Diensten belegt. Daraus ergibt sich, dass ungefähr ein Drittel der Dienste (vier von zwölf) verlegt werden müssen um einen Lastausgleich zu erzielen. Bei ca. 8000 Diensten müssen dementsprechend etwa 2660 Dienste verlegt werden. Diese sehr grobe Abschätzung zeigt, dass etwa doppelt so viele Dienste wie notwendig bei der einfachen Überstrategie verlegt werden.

Da jede Verlegung eines Dienstes mit Kosten verbunden ist, muss die Anzahl der Dienstverlegungen möglichst weit reduziert werden, ohne das Optimierungsergebnis stark zu verschlechtern. Das bedeutet, dass eine Übergabestrategie gefunden werden muss, die nur dann eine Verlegung eines Dienstes erlaubt, wenn der dadurch erzielte Optimierungsgewinn der Gesamtoptimierung dient.

Die Übergabestrategie mit adaptiver Schranke versucht sich anhand der gemessenen Lastunterschiede zwischen zwei Knoten ein Bild über das Optimierungspotential des Systems zu machen. Je größer die Lastunterschiede zwischen den Knoten sind, desto mehr Verlegungen sollen durchgeführt werden, während ein geringer Lastunterschied zwischen den Knoten dazu führt, dass die Verlegungen der Dienste unterbunden werden, da der zu erzielende Gewinn für die Optimierung keinen markanten Anteil liefert.

Gewinn

Zu diesem Zweck berechnet jeder Knoten den Gewinn $gain(s)$, der durch die Verlegung des Dienstes s erzielt werden kann und errechnet daraus den durchschnittlichen Gewinn $gain(k)$ seiner Verlegungen. Für die Verlegung eines Dienstes ist nun entscheidend, ob der Gewinn größer ist als die adaptive Schranke $barrier$.

$$gain(s) \geq barrier \tag{5.12}$$

Der Gewinn eines Dienstes entspricht der Last des zur Verlegung ausgewählten Dienstes, der nach dem gleichen Verfahren wie bei der gewichteten und einfachen Übergabestrategie ermittelt wird (siehe Abschnitt 5.3.1 Formel 5.5).

$$gain(s) = sload_{\text{avg}}(s) \tag{5.13}$$

Der durchschnittliche Gewinn eines Knotens k berechnet sich aus dem Durchschnitt der einzelnen Gewinne.

$$gain(k) = \frac{1}{n} \sum_{i=1}^{n} gain(s_{\text{i}}) \tag{5.14}$$

Die Gewinne nach Formel 5.13 und 5.14 einer Verlegung können nur berechnet werden, wenn auch ein Dienst für eine Verlegung ermittelt werden kann. Für den Fall, dass kein Dienst für eine Verlegung gefunden wird, weil beispielsweise der Senderknoten bereits eine höhere Last hat als der Empfängerknoten, wird der Gewinn auf Null gesetzt und in Formel 5.14 nicht berücksichtigt.

Optimierungspotential

Ein weiterer Wert zur Berechnung der adaptiven Schranke beschreibt das vorhandene Optimierungspotential $opt(k)$ der Knoten. Es berechnet sich aus dem Durchschnitt der Gewinne der letzten m Kommunikationen, wobei hier auch Gewinne mit $gain(s) = 0$ berücksichtigt werden, die gerade den Fall ausdrücken, dass kein Optimierungspotential vorhanden ist und somit die adaptive Schranke erhöhen soll, um Verlegungen zu unterdrücken.

$$opt(k) = \frac{1}{m} \sum_{i=1}^{m} gain(s_{\text{i}}) \tag{5.15}$$

Mit den Formeln 5.14 und 5.15 berechnet jeder Knoten seinen durchschnittlichen Gewinn über alle Nachrichten und seine Einschätzung des Optimierungspotentials aufgrund der letzten m Nachrichten. Aus diesen beiden Werten kann der Wert für die Schranke berechnet werden.

Parameter für die Berechnung der adaptiven Schranke

Um den Wert der Schranke nicht nur von den eigenen Werten abhängig zu machen, die gerade zu Beginn der Optimierung weit vom Optimum entfernt sein können, werden zusätzlich noch die Informationen des Sendeknotens in die Schrankenberechnung mit einbezogen. Zu diesem Zweck werden die beiden Werte $\Delta g(k_{\text{rec}})$ und $\Delta o(k_{\text{receive}})$ definiert.

Der Wert $\Delta g(k_{\text{rec}})$ beinhaltet den zeitlichen Verlauf der Gewinne der Senderknoten (k_{send}) und des Empfängerknotens (k_{rec}), jedoch nur wenn $gain(k) > 0$ ist.

$$\Delta g_{\text{neu}}(k_{\text{rec}}) = \frac{1}{2} gain(k_{\text{rec}}) + \frac{1}{4}(\Delta g_{\text{alt}}(k_{\text{rec}}) + gain(k_{\text{send}})) \tag{5.16}$$

Der Wert für $\Delta g_{\text{neu}}(k_{\text{rec}})$ berechnet sich zur Hälfte aus dem aktuellen durchschnittlichen Gewinn und zu je einem Viertel aus dem letzten Wert und dem aktuellen Gewinn des Sendeknotens. Der Wert kann nach Formel 5.16 nur dann berechnet werden, wenn alle Informationen verfügbar sind. Für den Fall, dass noch nicht alle Werte definiert sind, wird $\Delta g_{\text{neu}}(k_{\text{rec}})$ mit den Formeln 5.17 und 5.18 berechnet, abhängig davon, wie viele Werte bereits definiert sind.

$$\Delta g_{\text{neu}}(k_{\text{rec}}) = \begin{cases} \frac{1}{2}(gain(k_{\text{rec}}) + \Delta g_{\text{alt}}(k_{\text{rec}})) & \text{falls } gain(k_{\text{send}}) = N/A \\ \frac{1}{2}(gain(k_{\text{rec}}) + gain(k_{\text{send}})) & \text{falls } \Delta g_{\text{alt}}(k_{\text{rec}}) = N/A \\ \frac{1}{2}(\Delta g_{\text{alt}}(k_{\text{rec}}) + gain(k_{\text{send}})) & \text{falls } gain(k_{\text{rec}}) = N/A \end{cases} \tag{5.17}$$

$$\Delta g_{\text{neu}}(k_{\text{rec}}) = \begin{cases} gain(k_{\text{rec}}) & \text{falls } \Delta g_{\text{alt}}(k_{\text{rec}}) = N/A \text{ und } gain(k_{\text{send}}) = N/A \\ gain(k_{\text{rec}}) & \text{falls } gain(k_{\text{send}}) = N/A \text{ und } \Delta g_{\text{alt}}(k_{\text{rec}}) = N/A \\ \Delta g_{\text{alt}}(k_{\text{rec}}) & \text{falls } gain(k_{\text{send}}) = N/A \text{ und } gain(k_{\text{rec}}) = N/A \end{cases} \tag{5.18}$$

Der Wert $\Delta o(k_{\text{rec}})$ beschreibt das vorhandene Optimierungspotential in Abhängigkeit von dem aktuell berechneten $opt(k_{\text{rec}})$, dem vorherigen Optimierungspotential $\Delta o_{\text{alt}}(k_{\text{receive}})$ und dem des Senderknotens $opt(k_{\text{send}})$.

$$\Delta o_{\text{neu}}(k_{\text{rec}}) = \frac{1}{2} opt(k_{\text{rec}}) + \frac{1}{4}(\Delta o_{\text{alt}}(k_{\text{rec}}) + opt(k_{\text{send}})) \tag{5.19}$$

Da in die Berechnung des Optimierungspotentials $opt(k_{\text{rec}})$ auch Werte mit $gain(k) = 0$ einfließen, ist dieser auf jeden Fall definiert. Für den Fall, dass einer der beiden anderen Werte undefiniert ist, wird $\Delta o_{\text{neu}}(k_{\text{rec}})$ wie folgt berechnet:

$$\Delta o_{\text{neu}}(k_{\text{rec}}) = \begin{cases} \frac{1}{2}(opt(k_{\text{rec}}) + \Delta o_{\text{alt}}(k_{\text{rec}})) & \text{falls } opt(k_{\text{send}}) = N/A \\ \frac{1}{2}(opt(k_{\text{rec}}) + opt(k_{\text{send}})) & \text{falls } \Delta o_{\text{alt}}(k_{\text{rec}}) = N/A \\ opt(k_{\text{rec}}) & \text{falls } opt(k_{\text{send}}) = N/A \text{ und} \\ & \quad \Delta o_{\text{alt}}(k_{\text{rec}}) = N/A \end{cases} \tag{5.20}$$

Für die Berechnung der adaptiven Schranke *barrier* werden mit den Werten für $\Delta g(k_{\text{rec}})$ und $\Delta o(k_{\text{rec}})$ drei unterschiedliche Methoden getestet. Allen Varianten ist gemein, dass die adaptive Schranke *barrier* bei einer steigenden Tendenz des Optimierungspotentials fällt und bei niedrigem Optimierungspotential wächst.

Schranke mit linearem Adaptionsverhalten

Die intuitivste Variante stellt die lineare Abhängigkeit der Schranke $barrier_{\text{lin}}$ von beiden Werten dar.

$$barrier_{\text{lin}} = \Delta g(k_{\text{rec}}) - \Delta o(k_{\text{rec}}) \tag{5.21}$$

Dabei wird die Größe der Optimierungstendenz von dem durchschnittlichen Gewinn abgezogen. Befindet sich das System in einem gut balancierten Zustand, ist die Optimierungstendenz bei Null und damit der Wert der Schranke so groß wie der durchschnittliche Gewinn. Das bedeutet, dass ein Dienst nur dann verlegt wird, wenn der dadurch erzielte Gewinn den bisherigen durchschnittlichen Gewinn übersteigt. Ist die Optimierungstendenz hingegen groß, so fällt der Schrankenwert gegen Null. Somit können bei einer hohen Optimierungstendenz auch Dienste verlegt werden, deren Gewinn unter dem mittleren Gewinn liegt.

Schranke mit exponentiellem Adaptionsverhalten

Um ein schnelleres Adaptionsverhalten der Schranke zu erzielen wird ein exponentieller Anteil zur Adaption der Schranke benutzt. Für kleine Werte der Optimierungstendenz $\Delta o(k_{\text{rec}})$ wird der Exponent größer und somit die Schranke schnell ansteigen. Für den Fall, dass das Optimierungspontential den bisherigen durchschnittlichen Gewinn übersteigt, wird der Exponent negativ und die Schranke fällt gegen Null.

$$barrier_{\text{exp}} = \Delta g(k_{\text{rec}}) \cdot e^{\Delta g(k_{\text{rec}}) - \Delta o(k_{\text{rec}})} \tag{5.22}$$

Bedingt durch die Exponentialfunktion fällt der Anstieg der Schranke im positiven Bereich des Exponenten deutlich stärker aus als der Abfall im negativen Bereich des Exponenten. Daraus ergibt sich für die Schranke ein stärker hemmendes Verhalten.

Schranke mit logarithmischem Adaptionsverhalten

Die dritte Variante der adaptiven Schranke beinhaltet ein logarithmisches Adaptionsverhalten. Der Vorteil liegt in dem Verlauf der Logarithmusfunktion, die in dem Bereich zwischen 1 und 2 ein sehr sensibles Verhalten gegenüber kleinen Änderungen zeigt.

$$barrier_{\log} = \log_2\left(1 + \frac{\Delta g(k_{\text{rec}})}{\Delta o(k_{\text{rec}})}\right) \cdot \Delta g(k_{\text{rec}}) \tag{5.23}$$

Für die Berechnung der Schranke wird durch die Addition von 1 zum Quotienten aus dem durchschnittlichen Gewinn und der Optimierungstendenz, nur der positive Bereich der Logarithmusfunktion benutzt. Da die Logarithmusfunktion in diesem Bereich stark auf kleine Änderungen reagiert, wird ein sehr schnelles Adaptionsverhalten erzielt. Die Schranke mit logarithmischem Adaptionsverhalten eignet sich damit besonders gut für Systeme in denen sehr schnell auf Änderungen in den Lastverhältnissen reagiert werden muss.

Evaluierung der Übergabestrategien mit adaptiver Schranke

Die drei Übergabestrategien mit adaptiver Schranke werden mit den in Tabelle 5.1 angegebenen Parametern simuliert und der einfachen Übergabestrategie gegenübergestellt. Als zusätzlichen Parameter für die adaptive Schranke wird der Wert Δ_x angegeben, der die Anzahl der zu betrachtenden Werte angibt, die für die Berechnung der Optimierungstendenz benutzt werden. Die folgende Tabelle enthält die Simulationsergebnisse nach 2000 Simulationsschritten bei der entsprechenden Anzahl der zu betrachtenden Werte für die Optimierungstendenz. Die erste Zeile zeigt zum Vergleich die Werte der einfachen Übergabestrategie.

Alle drei Übergabestrategien zeigen in Bezug auf den initialen Wert der absoluten mittleren Abweichung $\overline{\delta_{nload_{\text{avg}}}}$ von 22,95 (siehe Tabelle 5.2) eine gute Balancierung der Lasten, benötigen dazu jedoch 20 bis 45 Prozent weniger Übergaben als die einfache Übergabestrategie. In Abbildung 5.4 ist der Zusammenhang zwischen der mittleren Abweichung der Knotenlast und der Anzahl der benötigten Übergaben vergleichend für die drei Übergabestrategien mit adaptiver Schranke und der einfachen Übergabestrategie dargestellt. Es ist zu erkennen, dass die Übergabestrategien mit adaptiver Schranke bei deutlich weniger Übergaben die Last nahezu gleich gut verteilen, wie die einfache Übergabestrategie.

Die Übergabestrategien mit adaptiver Schranke sind durchaus in der Lage, den guten Optimierungswert der einfachen Übergabestrategie zu erreichen, benötigen dazu jedoch deutlich mehr Simulationsschritte, weil eine Vielzahl an möglichen

Tabelle 5.5: Übergabestrategie mit linearem Adaptionsverhalten

Schranke	Δ_x	$\bar{t}$	σ_t	$\overline{\delta_{nload_{\text{avg}}}}$	$\sigma_{\delta_{nload_{\text{avg}}}}$
ohne	-	5313,65	95,35	0,66	0,02
lin	1	3681,28	52,95	0,81	0,02
	3	3724,08	62,36	0,81	0,02
	5	3792,66	66,25	0,82	0,02
	10	3893,43	62,89	0,81	0,02
exp	1	3341,19	56,35	1,30	0,03
	3	3075,56	49,37	1,49	0,04
	5	3016,09	49,29	1,54	0,05
	10	2977,13	50,18	1,57	0,06
log	1	3015,63	47,82	1,03	0,03
	3	2958,96	54,01	1,05	0,03
	5	2965,55	52,42	1,05	0,03
	10	2997,20	48,41	1,06	0,03

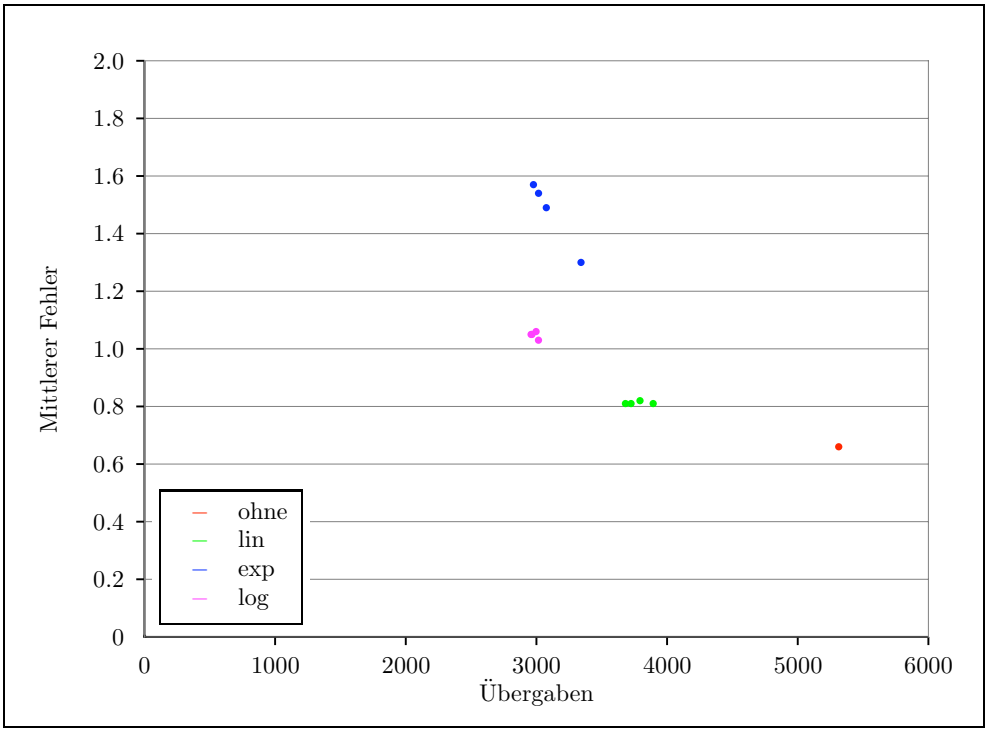

Abbildung 5.4: Vergleich der Übergabestrategien mit linearem (lin), mit exponentiellem (exp), mit logarithmischem (log) Adaptionsverhalten und der Übergabestrategie ohne Schranke

Übergaben durch die adaptive Schranke unterdrückt werden, da sie keinen signifikanten Anteil zur Optimierung liefern.

Der Unterschied zwischen den einzelnen Übergabestrategien mit adaptiver Schranke liegt in der Reaktion der Schranke auf Änderungen der Optimierungstendenz. Die Schranke mit linearem Adaptionsverhalten reagiert sehr schnell auf Änderungen und erlaubt dadurch mehr Übergaben als die beiden anderen Verfahren. Das exponentielle Adaptionsverhalten führt zu einer eher hemmenden Wirkung in Bezug auf die Übergaben, da die Schranke im positiven Bereich deutlich schneller wächst als im negativen Bereich. Die logarithmische Variante bietet aufgrund der hohen Adaptionsgeschwindigkeit der Schranke in beiden Richtungen einen guten Kompromiss zwischen der Anzahl der Übergaben und der zu erzielenden Güte.

5.3.4 Übergabestrategie mit Lastschätzer

Die Ergebnisse der Übergabestrategie mit logarithmischem Adaptionsverhalten der Schranke zeigen bereits eine deutliche Reduzierung der notwendigen Übergaben im Vergleich zur einfachen Übergabestrategie. Es werden ca. 45 Prozent weniger Übergaben benötigt, um den mittleren Fehler von 1,05 zu erzielen. Im Fall der Schranke mit linearem Adaptionsverhalten kann sogar ein mittlerer Fehler von 0,81 erreicht werden, wobei ca. 20 Prozent weniger Übergaben notwendig sind.

Alle drei Verfahren benötigen jedoch immer noch entweder mehr Übergaben, als im theoretischen Fall in Kapitel 5.3.3 abgeschätzt wurde, oder erreichen nicht den Optimierungswert von 0,66 der einfachen Übergabestrategie. Das Problem liegt darin, dass der erzielte Gewinn einer Übergabe zwar einen markanten Anteil zur Gesamtoptimierung beiträgt, da die Schranke Verlegungen mit kleinen Gewinnen unterdrückt, aber die Verlegungen nicht zwangsläufig die Last der betroffenen Knoten an die mittlere Last des Gesamtsystems annähert. Werden also zwei Knoten mit unterschiedlichen Lasten gefunden und auch ein Dienst, der die Last zwischen den beiden Knoten ausgleicht, kann dies trotzdem zur Folge haben, dass der Empfängerknoten, der einen Dienst zum Senderknoten verlegt, entlastet wird und damit der Senderknoten, der bereits über der mittleren Last liegt noch weiter davon entfernt wird. Das gleiche Problem kann in umgekehrter Weise auftreten, wenn die Last der beiden Knoten unterhalb der mittleren Last des Systems liegt.

Die dargestellten Probleme entstehen aus Mangel an Wissen über die tatsächliche Last des Gesamtsystems. Da die Selbstorganisationsstrategie völlig ohne zentrale Kontrolle auskommen soll, muss ein Weg gefunden werden, die mittlere Last des Systems zu ermitteln, ohne die Information von allen Knoten zu erfragen. Hierzu soll ein Lastschätzer zum Einsatz kommen, der anhand der Lasten anderer Knoten

eine Abschätzung über die Gesamtlast des Systems ermittelt. Der Lastschätzer $\Delta load$ wird aus der Last des eigenen Knotens (siehe Formel 5.3) und den letzten m Schätzwerten, die von anderen Knoten empfangen wurden, berechnet.

$$\Delta load(n_{\text{rec}}) = \begin{cases} nload_{\text{avg}}(k_{\text{rec}}) & \text{falls } m = 0 \\ \frac{1}{m+1}\left(nload_{\text{avg}}(k_{\text{rec}}) + \sum_{i=1}^{m} \Delta load(n_{\text{send}}(i)) \right) & \text{falls } m > 0 \end{cases} \tag{5.24}$$

Für die Übergabestrategie mit Lastschätzer sind nun zwei Parameter von Interesse, die die Güte der Strategie beeinflussen. Zum einen kann der Wert für die Länge der Schätzwertliste m, also die Anzahl der gespeicherten Schätzwerte variiert werden, zum anderen existiert der Wert *start*, der angibt, nach wie vielen empfangenen Schätzwerten von anderen Knoten eine Verlegung eines Dienstes zum ersten Mal erlaubt wird.

Die Metrik erlaubt die Verlegung eines Dienstes nur dann, wenn die Last des Empfängerknotens über und die Last des Senderknotens unterhalb von $\Delta load$ liegt. Nur in diesem Fall ist sichergestellt, dass beide Knoten auf das Gesamtoptimum des Systems hin verändert werden, da normalerweise bei beiden Knoten die Lasten in Richtung des Schätzwertes geändert werden. Für den Fall, dass die Last des Senderknotens zwar kleiner ist als $\Delta load$, aber sehr nahe bei diesem Wert liegt, kann es vorkommen, dass durch eine Verlegung eines Dienstes die Last des Senderknotens über den Schätzwert hinaus erhöht wird.

Evaluierung der Übergabestrategien mit Lastschätzer

Tabelle 5.6 zeigt die Ergebnisse aus jeweils 100 Simulationen nach 2000 Simulationsschritten für die Werte von fünf bis zehn des Parameters m, der die Anzahl der empfangenen Schätzwerte anderer Knoten angibt. Der Parameter *start* war bei alle Simulationen zehn. Das bedeutet, dass ein Dienst erst dann verlegt werden kann, wenn mindestens zehn Schätzwerte von anderen Knoten empfangen wurden.

Den Verlauf der Übergaben zeigt das rechte Diagramm in Abbildung 5.5. Alle Kurven zeigen einen ähnlichen Verlauf bis etwa 100 Simulationsschritte. Anschließend wird der Verlauf der Kurve mit zunehmender Länge der Schätzwertliste deutlich flacher und die Anzahl der verlagerten Dienste nimmt ab. Für die Länge der Schätzwertliste von fünf ist deutlich zu erkennen, dass nach ca. 700 Übergaben nur noch eine geringe Anzahl an Diensten verlegt wird. Bei einer Länge von zehn hingegen scheint das Ende der Verlagerungen noch nicht erreicht, was daran liegt, dass in die Berechnung des Lastschätzers auch Werte einbezogen werden, die sehr

Tabelle 5.6: Übergabestrategie mit Lastschätzer

m	$\bar{t}$	σ_t	$\overline{\delta_{nload_{\text{avg}}}}$	$\sigma_{\delta_{nload_{\text{avg}}}}$
5	2559,80	43,53	0,64	0,02
6	2448,42	34,76	0,64	0,02
7	2377,52	39,29	0,64	0,02
8	2333,53	34,96	0,63	0,02
9	2285,52	33,77	0,64	0,02
10	2193,21	34,64	0,70	0,02

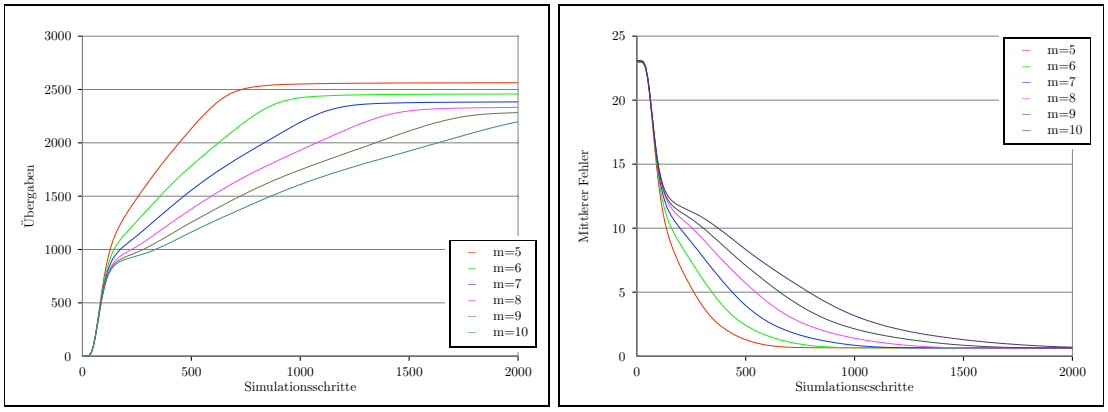

Abbildung 5.5: Anzahl der Dienstverlegungen $\bar{t}$ (links) und mittlerer Fehler $\overline{\delta_{nload_{\text{avg}}}}$ (rechts) bei der Übergabestrategie mit Lastschätzer

weit zurückliegen und darum der tatsächliche mittlere Lastwert später erreicht wird. Tabelle 5.6 zeigt, dass bei einer Länge von acht ein guter Kompromiss zwischen der Anzahl der Übergaben und dem zu erzielenden mittleren Fehler erreicht wird.

Das rechte Diagramm in Abbildung 5.5 zeigt den Verlauf des mittleren Fehlers für die verschiedenen Längen der Schätzwertliste. Mit zunehmender Länge wird der beste Wert von ca. 0,64 langsamer erreicht und bei zehn, aufgrund der noch nicht abgeschlossenen Übergaben sogar nur 0,70.

5.3.5 Übergabestrategie mit Hybridverfahren

Die vorausgegangenen Evaluierungen zeigen, dass die Übergabestrategie mit Lastschätzer das beste Ergebnis in Bezug auf die notwendigen Übergaben und den dadurch erzielten mittleren Fehler liefert und die Übergabestrategie mit adaptiver Schranke die Übergaben von Diensten unterdrückt, die nur einen geringen Anteil zur Optimierung beitragen.

Darüber hinaus haben Evaluierungen mit dynamischem Prozessverhalten gezeigt, dass die Übergabestrategien mit Lastschätzer, die einfache Übergabestrategie sowie die gewichtete Übergabestrategie bei wechselnden Lasten zu oszillieren beginnen. Die Übergabestrategie mit adaptiver Schranke verhindert nicht nur, dass Verlegungen unterhalb eines gewissen Gewinns unterdrückt werden, sondern auch ein Oszillieren des Systems im Fall von dynamischen Laständerungen, da sich die adaptive Schranke an das mittlere Optimierungspotential anpasst. Der Nachteil der Übergabestrategie mit adaptiver Schranke liegt in dem schlechteren mittleren Fehler, der jedoch dadurch zustande kommt, dass diese Strategie den mittleren Lastwert des Netzes nicht kennt.

Die Übergabestrategie mit Hybridverfahren versucht die Vorteile der beiden Übergabestrategien zu vereinen, um mit wenigen Übergaben einen möglichst kleinen mittleren Fehler zu erzielen und somit die Last im Netz optimal zu erteilen. Der Lastschätzer soll wie bisher auch die mittlere Last im Netz ermitteln und die adaptive Schranke den mittleren Gewinn. Im Fall von dynamischem Prozessverhalten soll die adaptive Schranke Oszillationen im Netz verhindern.

Das Hybridverfahren arbeitet in zwei Stufen. Zunächst versucht der Lastschätzer möglichst schnell das Netz zu optimieren. Anschließend wird die adaptive Schranke benutzt, um Übergaben mit geringem Gewinn zu unterdrücken. Die Übergabestrategie benötigt somit ein Entscheidungskriterium, um die adaptive Schranke zu aktivieren.

Zu diesem Zweck wird die Veränderung des Schätzwertes betrachtet, um die Unterschiede der Knoten bewerten zu können. Der Schätzwert des Lastschätzers wird aus dem arithmetischen Mittel der letzten m Werte der empfangenen Schätzwerte anderer Knoten und dem eigenen Schätzwert berechnet. Ändert sich dieser Wert um weniger als 10 Prozent gegenüber dem letzten Wert wird die adaptive Schranke in die Bewertung der Übergaben mit einbezogen. Formel 5.25 gibt die Berechnung zweier aufeinander folgender Lastdifferenzen an.

$$load_{\text{diff}} = \Delta load_t - \Delta load_{t-1} \tag{5.25}$$

Die adaptive Schranke berechnet von Beginn an ihren Schrankenwert, wird jedoch solange für die Beurteilung einer Übergabe nicht benutzt, solange die Änderungen zwischen zwei aufeinander folgenden Lastschätzwerten mehr als 10 Prozent in Bezug auf den neuen Schätzwert beträgt. Sobald die adaptive Schranke benutzt wird, wird zunächst mit dem Lastschätzer beurteilt, ob die beiden betroffenen Knoten für eine Übergabe in Frage kommen und anschließend durch die adaptive Schranke der Gewinn ermittelt. Liegt dieser über der Schranke wird die Übergabe durchgeführt, andernfalls unterdrückt. Algorithmus 4 gibt die Entscheidungsfindung der Übergabestrategie mit Hybridverfahren in Form von Pseudo-Code an.

Algorithmus 4 Entscheidung der Übergabestrategie mit Hybridverfahren

1: $\Delta load_{t-1} = \Delta load_t$
2: calulate $\Delta load_t$
3: $load_{\text{diff}} = \Delta load_t - \Delta load_{t-1}$
4: **if** $\Delta load(n_{\text{rec}}) > \Delta load_t$ and $\Delta load(n_{\text{rec}}) < \Delta load_t$ **then**
5: find best service for transfer
6: **if** $load_{\text{diff}} < 0.1 * \Delta load_t$ **then**
7: **if** $gain > barrier$ **then**
8: transfer service
9: **end if**
10: **else**
11: transfer service
12: **end if**
13: **end if**

Evaluierung der Übergabestrategie mit Hybridverfahren

Die Übergabestrategie mit Hybridverfahren kombiniert die Verfahren der Übergabestrategie mit Lastschätzer und der Übergabestrategie mit adaptiver Schranke. Aus diesem Grund wird die Übergabestrategie mit Hybridverfahren mit variierenden Werten für die Länge der Schätzwertliste m und der Anzahl der zu betrachtenden Werte der Optimierungstendenz Δ_x evaluiert. Die Ergebnisse sind in Tabelle 5.7 dargestellt.

Für die Längen der Schätzwertliste m von sieben bis neun variiert die Anzahl der Übergaben innerhalb der Größe der Standardabweichung unabhängig von Δ_x. Daraus kann abgeleitet werden, dass diese Parametereinstellungen nahezu gleich gute Ergebnisse liefern.

Der mittlere Fehler liegt mit durchschnittlich 1,44 über dem der anderen Verfahren. Im Gegenzug werden lediglich 75 Prozent der Übergaben im Vergleich zur Übergabestrategie mit Lastschätzer und der Anzahl der Übergaben im theoretischen Fall benötigt.

Wieder ist zu erkennen, dass für eine zu lange Schätzwertliste keine gute Optimierung innerhalb der vorgegebenen Schrittzahl erreicht werden kann, da zu viele Verlegungen unterdrückt werden.

Die Anzahl der betrachteten Werte der Optimierungstendenz Δ_x scheint eine untergeordnete Rolle zu spielen, da die Ergebnisse nur um einen geringen Wert schwanken. Einen etwas stärkeren Einfluss scheint die Länge der Schätzwertliste zu haben, da hierdurch besonders bei den Werten fünf und sechs sehr große Veränderungen beobachtet werden können. Bei den Werten sieben, acht und neun sind die Änderungen zwar nicht mehr so markant, jedoch größer als die Auswirkungen des Parameters Δ_x

Tabelle 5.7: Übergabestrategie mit Hybridverfahren

m	Δ_x	$\bar{t}$	σ_t	$\overline{\delta_{nload_{\text{avg}}}}$	$\sigma_{\delta_{nload_{\text{avg}}}}$
5	3	1834,31	28,37	1,27	0,035
	5	1837,22	32,18	1,29	0,031
	10	1850,93	31,53	1,30	0,033
6	3	1803,75	32,86	1,32	0,038
	5	1791,73	32,62	1,33	0,039
	10	1807,77	33,08	1,34	0,039
7	3	1776,06	33,89	1,37	0,037
	5	1764,73	31,03	1,39	0,045
	10	1772,49	29,63	1,41	0,041
8	3	1754,91	28,28	1,44	0,041
	5	1747,96	28,86	1,45	0,042
	10	1755,48	33,04	1,48	0,045
9	3	1726,95	32,26	1,51	0,043
	5	1724,92	32,25	1,53	0,048
	10	1728,11	30,90	1,56	0,047
10	3	1716,71	29,70	1,59	0,046
	5	1711,82	27,98	1,63	0,052
	10	1709,90	31,66	1,66	0,053

Die großen Unterschiede in der Anzahl der Übergaben ist mit dem anfänglich schnelleren Optimierungsverhalten bei den Werten fünf und sechs für die Länge der Schätzwertliste zu erklären. Da der errechnete Schätzwert aus der Schätzwerthistorie alte Informationen früher überschreibt, wird zu Beginn ein besseres Adaptionsverhalten erzielt. Im weiteren Verlauf sind jedoch längere Schätzwertlisten von Vorteil, da sich Ausreißer in den Schätzwerten weniger stark auswirken.

Die besten Ergebnisse werden bei den Werten sieben und acht für die Länge der Schätzwertliste m und $\Delta_x = 3$ erreicht. Dies bestätigt die Ergebnisse aus den Abschnitten 5.3.3 und 5.3.4, bei denen mit den jeweiligen Parametern die besten Ergebnisse ermittelt wurden. Daraus kann abgeleitet werden, dass die beiden Einzelverfahren, die in der Übergabestrategie mit Hybridverfahren zum Einsatz kommen, wie gedacht, unabhängig voneinander die Optimierung positiv verstärken. Während die Übergabestrategie mit Lastschätzer die Verlegungen auswählt, die grundsätzlich in Frage kommen, wählt die Übergabestrategie mit adaptiver Schranke nur noch die Übergaben aus, die auch tatsächlich zu einer Optimierung beitragen, da ihr Gewinn über dem Wert der adaptiven Schranke liegt.

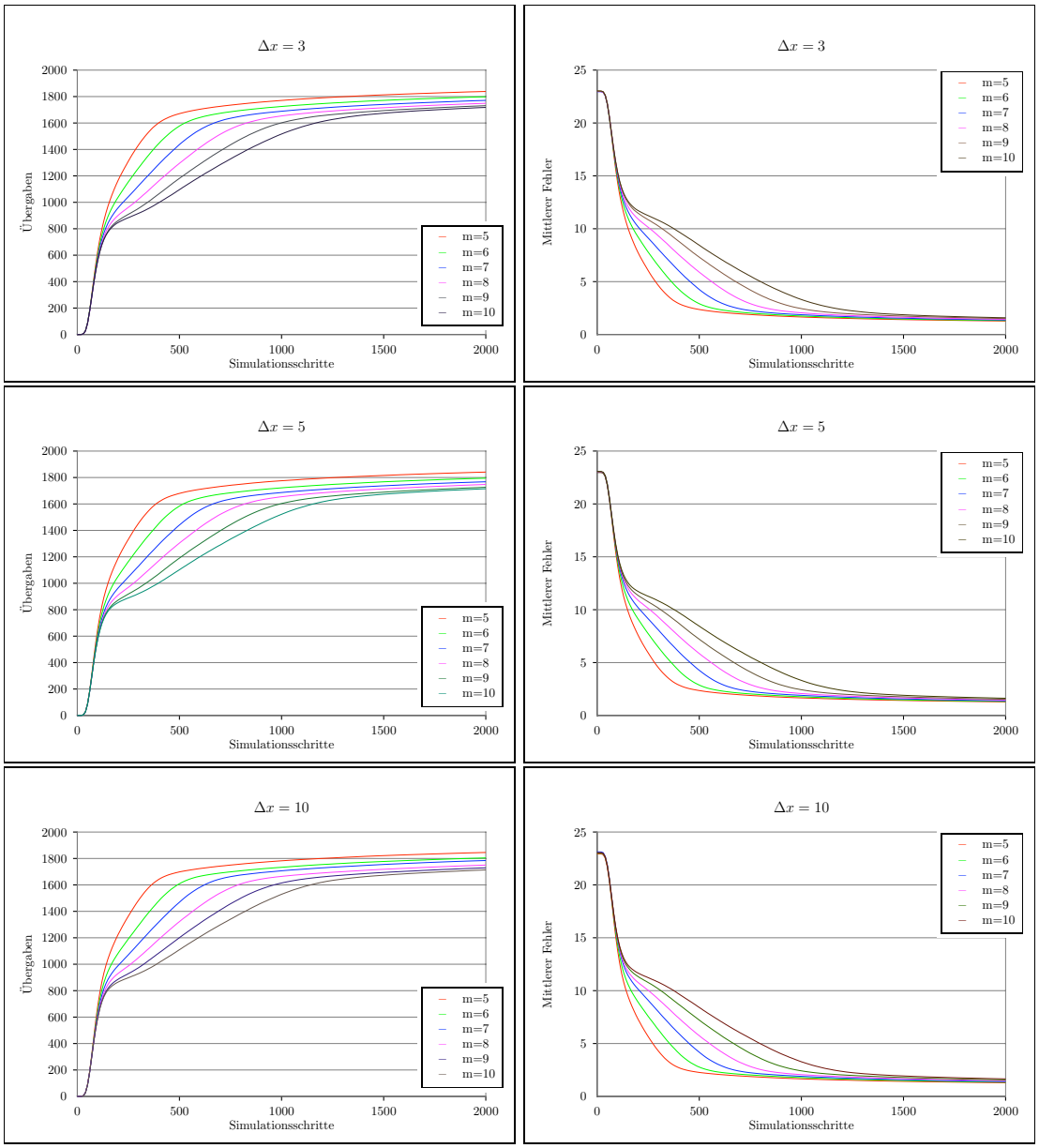

Abbildung 5.6: Anzahl der Dienstverlegungen $\bar{t}$ (links) und mittlerer Fehler $\overline{\delta_{nload_{\text{avg}}}}$ (rechts) bei der Übergabestrategie mit Hybridverfahren

Die Diagramme in Abbildung 5.6 zeigen die Ergebnisse der Simulationen mit 2000 Schritten. Links ist die Anzahl der Übergaben und rechts der mittlere Fehler abgebildet. Die Anzahl der Übergaben wird jeweils mit den Werten drei, fünf und zehn für die Anzahl der Optimierungstendenzwerte Δ_x dargestellt. In jedem Diagramm sind die Werte für die Längen der Schätzwertliste von fünf bis zehn abgebildet. Die Diagramme auf der rechten Seite zeigen jeweils die zugehörigen Messwerte für die mittleren Fehler bei gleichen Parametereinstellungen.

Die Diagramme zeigen keine offensichtlichen Unterschiede zwischen den einzelnen Werten von Δ_x. Lediglich die Geschwindigkeit der Optimierung unterscheidet sich aufgrund der Werte für m.

5.4 Variation der Simulationsparameter

In diesem Abschnitt sollen die Einflüsse der einzelnen Parameter auf die Simulationsergebnisse untersucht werden. Im Einzelnen werden die Auswirkungen bei Veränderung der Knotenanzahl, der durchschnittlichen Prozessgröße, der Kommunikationsrate und der Anzahl der durchschnittlichen Kommunikationspartner pro Knoten betrachtet.

5.4.1 Knotenanzahl

Die Anzahl der Knoten, die in einem Netzwerk vorhanden sind definieren zum einen die Größe des Netzwerkes, zum anderen die vorhandenen Ressourcen. Da in vielen Fällen der Aufwand eines Algorithmus mit der Anzahl der zu betrachtenden Elemente steigt, wird hier der Zusammenhang zwischen der Anzahl der Knoten und der damit verbundenen Anzahl an Diensten, sowie der daraus resultierende Aufwand für die Selbstoptimierung untersucht.

Der Simulator bestimmt zu Beginn einer Simulation für jeden Knoten den Wert $load_{\max}$ der die maximale Kapazität der vorhandenen Ressourcen beschreibt. Es werden dann so viele Dienste erzeugt, bis die Ressourcen weitestgehend erschöpft

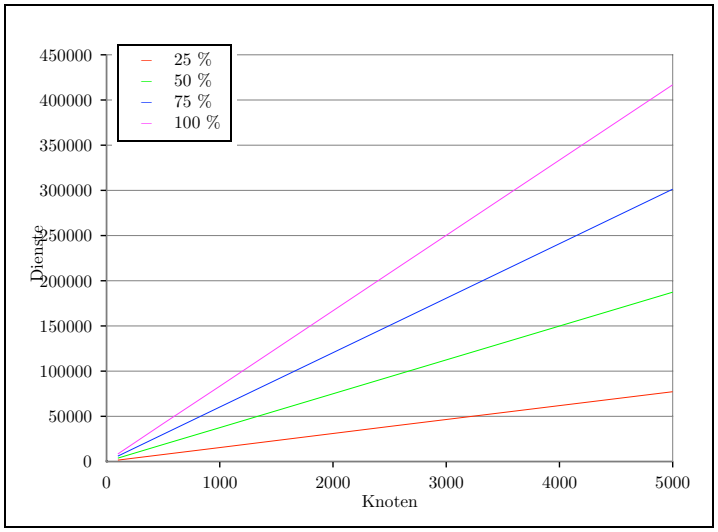

Abbildung 5.7: Anzahl der erzeugten Dienste in Abhängigkeit von der Anzahl der Knoten und der maximalen prozentualen Belastung der Knoten

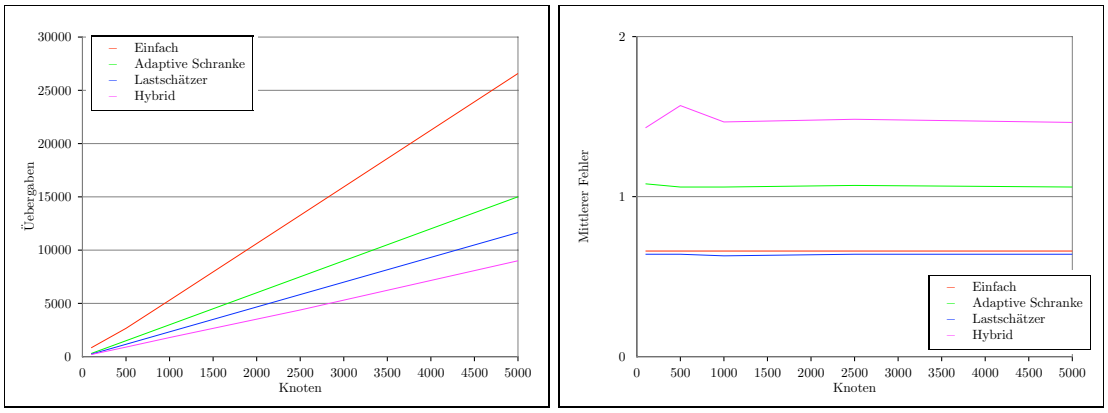

Abbildung 5.8: Anzahl der Übergaben und der mittlerer Fehler in Abhängigkeit von der Knotenzahl

sind. Wie Abbildung 5.7 zeigt, ist die Anzahl der erzeugten Dienste proportional zu der Anzahl der Knoten im Netzwerk. Die Abbildung zeigt die Anzahl der generierten Dienste in Abhängigkeit von der Anzahl der Knoten und der maximalen prozentualen Auslastung der Knoten.

Das linke Diagramm in Abbildung 5.8 zeigt die Anzahl der Übergaben in Abhängigkeit von der Knotenanzahl bei 2000 Simulationsschritten der unterschiedlichen Übergabestrategien. Es ist deutlich zu erkennen, dass die Selbstoptimierung für alle Übergabestrategien ein lineares Verhalten bezüglich der Anzahl der Übergaben zeigt. Dieses Verhalten ist besonders für große Netzwerke mit sehr vielen Knoten von Bedeutung, da Verfahren mit exponentiellem Aufwand dann nicht mehr anwendbar sind. Die Simulationsergebnisse zeigen, dass selbst in Netzwerken mit einer hohen Anzahl an Knoten und Diensten der Aufwand nur linear wächst.

Im rechten Diagramm in Abbildung 5.8 ist der mittlere Fehler in Abhängigkeit der Anzahl der Knoten abgebildet. Da der mittlere Fehler bei allen Netzgrößen gleich gering ist, skaliert das Verfahren nicht nur linear mit der Größe des Netzwerks, sondern erzielt dabei auch in allen Fällen ein gutes Ergebnis.

5.4.2 Prozessgröße

Der Simulator erzeugt während der Initialisierung für jeden Knoten so viele Dienste, dass die Ressourcen des Knotens so weit wie möglich erschöpft sind. Dabei liegt der Ressourcenverbrauch eines Dienstes innerhalb eines definierten Bereichs. Der Ressourcenverbrauch wird für jeden Dienst und jede Ressource zufällig gewählt, um ein möglichst breites Spektrum unterschiedlicher Dienste zu generieren. Der Parameter der Prozessgröße definiert dabei die obere Schranke dieses Bereichs.

Tabelle 5.8: Einfluss der Prozessgröße

%	Strategie	$\bar{t}$	σ_t	$\overline{\delta_{nload_{\text{avg}}}}$	$\sigma_{\delta_{nload_{\text{avg}}}}$
5	Einfach	9009,62	141,36	0,29	0,015
	Adaptiv	5565,39	94,60	0,58	0,016
	Lastschätzer	4180,76	69,17	0,27	0,013
	Hybrid	3469,45	62,25	0,86	0,027
10	Einfach	5307,98	81,16	0,66	0,022
	Adaptiv	2970,08	49,35	1,06	0,026
	Lastschätzer	2330,78	35,47	0,64	0,018
	Hybrid	1747,59	31,88	1,46	0,049
20	Einfach	3073,84	72,39	1,61	0,043
	Adaptiv	1657,78	28,05	2,15	0,063
	Lastschätzer	1428,27	28,52	1,56	0,046
	Hybrid	938,44	17,14	2,57	0,076
30	Einfach	2103,79	61,47	2,84	0,113
	Adaptiv	1169,91	28,14	3,47	0,102
	Lastschätzer	1122,47	29,38	2,89	0,114
	Hybrid	682,26	14,39	3,93	0,126
40	Einfach	1533,57	56,02	3,81	0,119
	Adaptiv	895,39	22,62	4,42	0,129
	Lastschätzer	933,75	24,06	3,79	0,124
	Hybrid	543,00	14,44	4,78	0,171
50	Einfach	956,40	78,72	5,77	0,410
	Adaptiv	622,70	30,73	6,08	0,419
	Lastschätzer	591,32	54,81	5,82	0,463
	Hybrid	408,42	11,03	6,18	0,327

Der Parameter der maximalen Prozessgröße bestimmt indirekt die Anzahl der generierten Dienste, da der Simulator versucht so viele Dienste zu erzeugen, bis die Ressourcen der Knoten erschöpft sind. Bei einem kleineren Wert für die maximale Prozessgröße werden somit mehr Dienste benötigt, bis die Ressourcen eines Knotens erschöpft sind, als bei einem größeren Wert der maximalen Prozessgröße.

Mit diesem Hintergrund sind die Simulationsergebnisse in Tabelle 5.8 entsprechend zu interpretieren. Die Anzahl der durchschnittlichen Übergaben $\bar{t}$ nimmt für alle Übergabestrategien mit zunehmender maximaler Prozessgröße ab, da weniger Dienste, jedoch mit größerem Ressourcenverbrauch, erzeugt werden und somit auch weniger Dienste zum Ausgleich zwischen den Knoten benötigt werden.

Für die Werte von fünf bis zwanzig für die maximale Prozessgröße zeigt die Übergabestrategie mit Hybridverfahren einen deutlich größeren mittleren Feh-

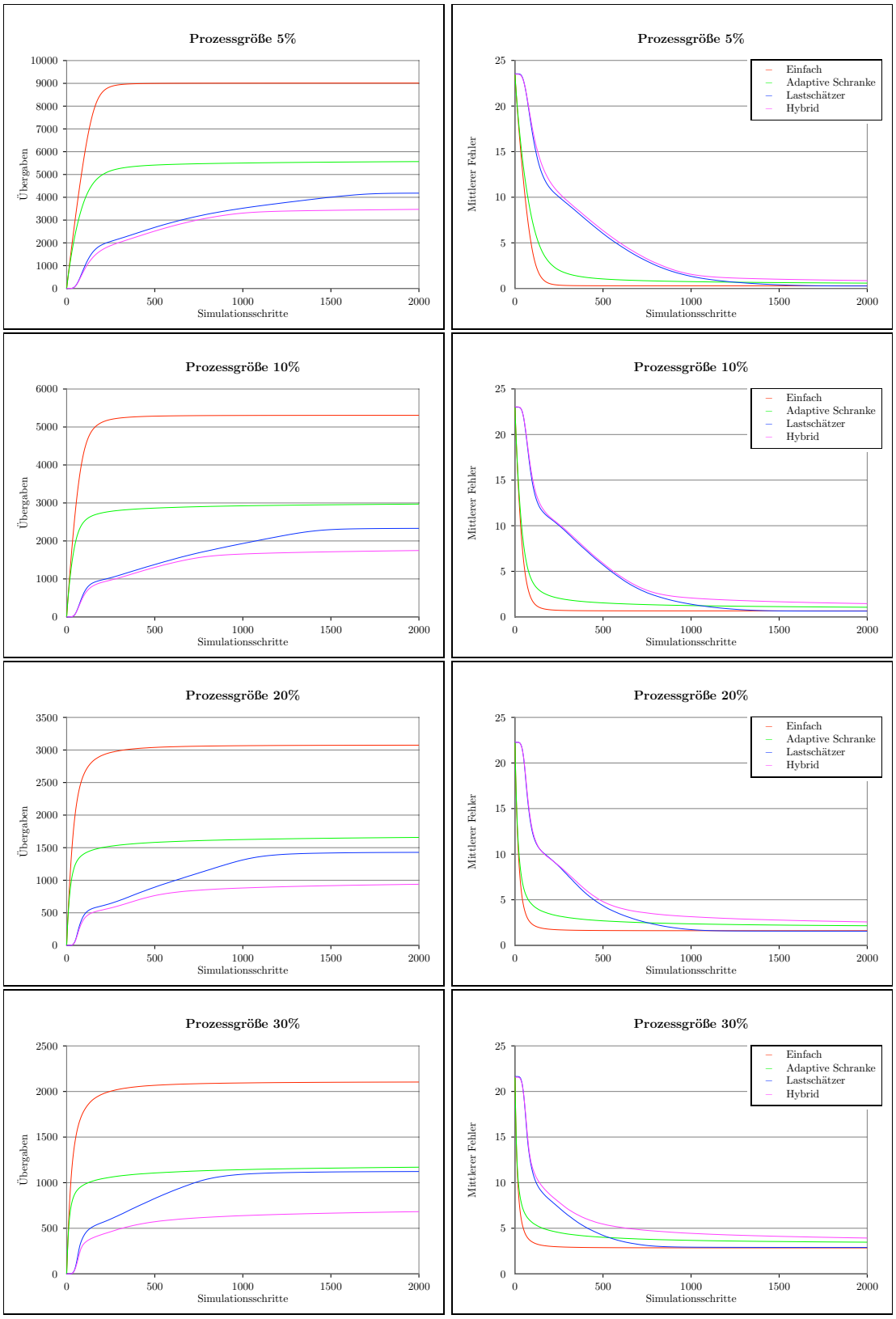

Abbildung 5.9: Anzahl der Übergaben in Abhängigkeit der Prozessgröße

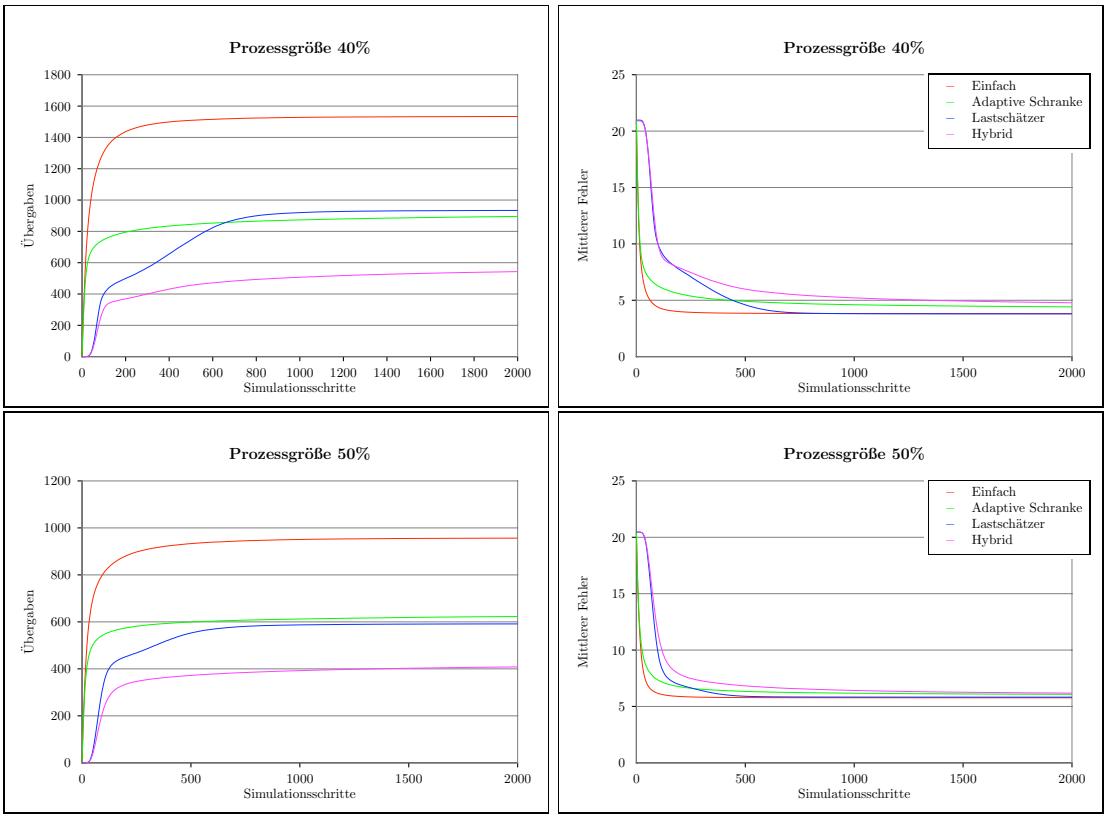

Abbildung 5.10: Anzahl der Übergaben in Abhängigkeit der Prozessgröße

ler als die anderen Übergabestrategien. Dieses Verhalten wird durch die adaptive Schranke erzeugt, die Übergaben mit einem Gewinn unterhalb der Schranke unterdrückt. Da hier jedoch gerade sehr viele Dienste mit kleinem Ressourcenverbrauch vorhanden sind, werden sehr viele Übergaben unterdrückt.

Für die maximalen Prozessgrößen von 30, 40 und 50 kehrt sich das Verhalten um. Bei 50 erzeugt die Übergabestrategie mit Hybridverfahren das beste Verhältnis zwischen der Anzahl der Übergaben und dem mittleren Fehler.

Mit zunehmender Prozessgröße steigt verständlicher Weise auch der mittlere Fehler, da nicht nur insgesamt weniger Dienste im System vorhanden sind, sondern zusätzlich auch weniger Dienste mit geringem Ressourcenverbrauch, die zu einem besseren Ausgleich des mittleren Fehlers herangezogen werden könnten.

Die Diagramme aus Abbildung 5.9 und 5.10 zeigen sehr anschaulich das Verhalten der einzelnen Übergabestrategien. Die Übergabestrategien mit adaptiver Schranke und Hybridverfahren unterdrücken bei einer maximalen Prozessgröße von fünf so viele Übergaben innerhalb der 2000 Simulationsschritte, so dass kein optimaler Endwert erreicht werden kann. Mit steigender maximaler Prozessgröße arbeiten beide Übergabestrategien deutlich besser. Entsprechendes gilt für den

mittleren Fehler, der sich bei den beiden Verfahren zunächst nur sehr langsam dem Optimalwert nähert.

5.4.3 Kommunikationsrate

Die Kommunikationsrate bestimmt die Anzahl der Knoten, die in einem Simulationsschritt eine Nachricht versenden und somit eine Information für die Selbstoptimierung übertragen. Eine Veränderung dieses Parameters kann auf zwei unterschiedliche Arten interpretiert werden. Zum einen kann eine höhere Kommunikationsrate bedeuten, dass in einem gegebenen Zeitraum mehr Nachrichten ausgetauscht werden, zum anderen kann es durch einen längeren Beobachtungszeitraum erklärt werden. Da in beiden Fällen eine höhere Anzahl an Nachrichten im gleichen Zeitintervall entsteht, können diese mit Variation der Kommunikationsrate simuliert werden.

Die Kommunikationsrate des Simulators definiert die Anzahl der Knoten, die bei einem Simulationsschritt als Sender einer Nachricht ausgewählt werden. Aus der Liste der Kommunikationspartner jedes Senders wird dann für jeden Knoten ein Empfänger der Nachricht ermittelt. Zehn Prozent bedeutet somit, dass zehn Prozent der Knoten des Netzwerks eine Nachricht senden.

Die Kommunikationsrate beeinflusst maßgeblich die Güte der Selbstoptimierung. Je weniger Nachrichten zwischen den Knoten übertragen werden, desto weniger

Tabelle 5.9: Einfluss der Kommunikationsrate

%	Strategie	$\overline{t}$	σ_t	$\overline{\delta_{nload_{\text{avg}}}}$	$\sigma_{\delta_{nload_{\text{avg}}}}$
5	Einfach	5326,51	100,11	0,66	0,02
	Adaptive Schranke	2918,40	49,25	1,24	0,03
	Lastschätzer	1862,35	32,84	1,55	0,07
	Hybrid	1632,00	26,72	2,16	0,06
10	Einfach	5307,70	93,48	0,66	0,02
	Adaptive Schranke	2969,46	53,56	1,06	0,03
	Lastschätzer	2336,65	35,44	0,64	0,02
	Hybrid	1754,91	27,28	1,43	0,04
25	Einfach	5256,56	92,22	0,66	0,02
	Adaptive Schranke	3010,87	50,73	0,91	0,02
	Lastschätzer	2410,44	40,33	0,64	0,02
	Hybrid	1881,48	26,44	1,02	0,03
50	Einfach	5159,94	92,26	0,65	0,02
	Adaptive Schranke	3029,20	51,33	0,84	0,02
	Lastschätzer	2512,57	42,90	0,64	0,02
	Hybrid	1978,17	32,12	0,87	0,03

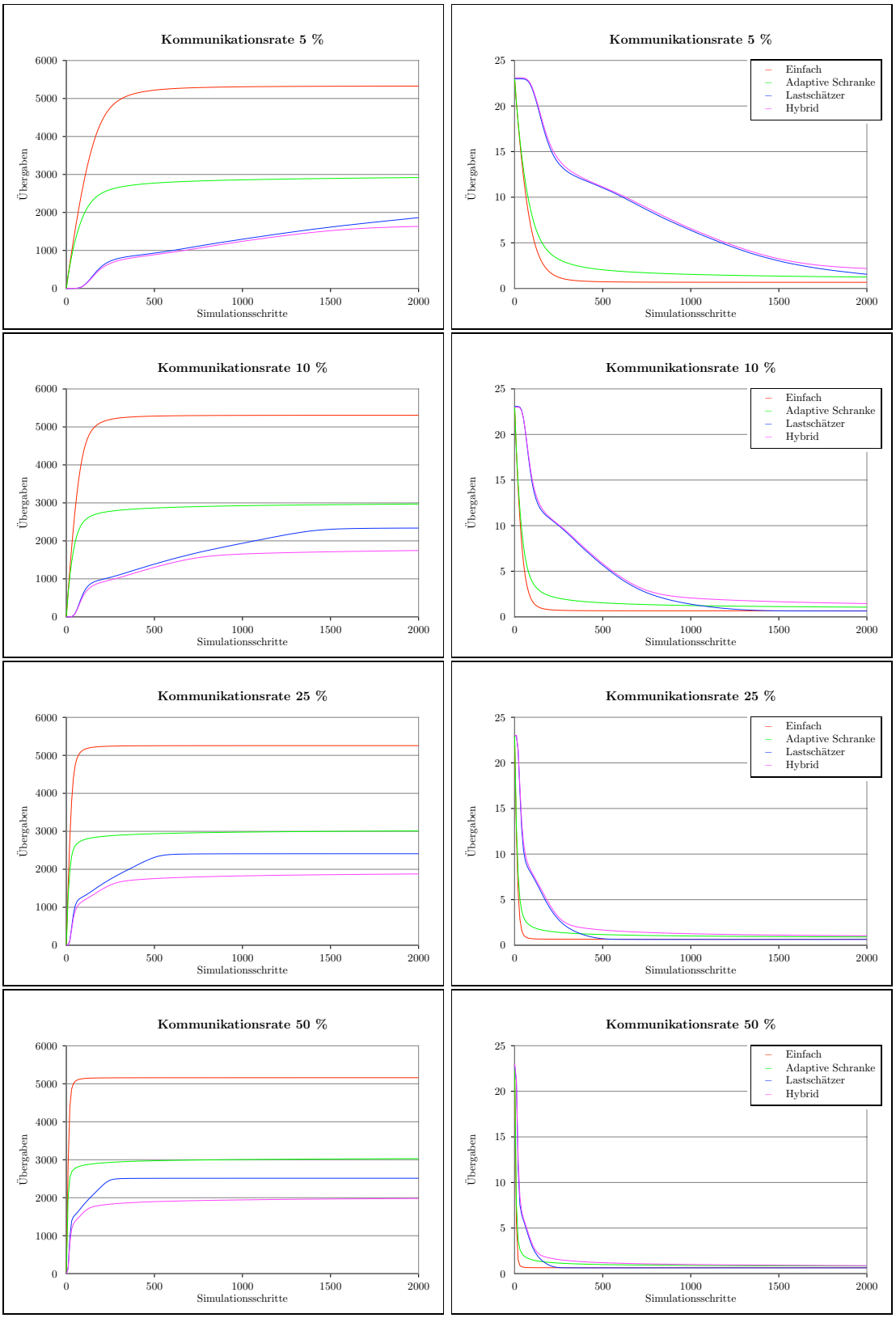

Abbildung 5.11: Anzahl der Übergaben in Abhängigkeit der Kommunikationsrate der Knoten

Information für die Selbstoptimierung kann zwischen den Knoten ausgetauscht werden. In Tabelle 5.9 sind die Ergebnisse für 5, 10, 25 und 50 Prozent der Kommunikationsrate dargestellt.

Die Diagramme aus Abbildung 5.11 zeigen das Verhalten der Übergabestrategien bei den unterschiedlichen Kommunikationsraten. Die linken Diagramme zeigen die Anzahl der Übergaben, die rechten den dabei erzielten mittleren Fehler.

Bei einer Kommunikationsrate von fünf Prozent sind die Übergabestrategien mit adaptiver Schranke und Hybridverfahren nicht in der Lage, das Netz zu optimieren, da offensichtlich nicht genügend Information verteilt wird, damit die beiden Übergabestrategien die notwendige Anzahl an Übergaben für die Optimierung finden können. Dementsprechend langsam nähert sich der mittlere Fehler seinem Endwert nach 2000 Schritten. Ab einer Kommunikationsrate von zehn Prozent erreichen die beiden Übergabestrategien auch wieder den Sättigungsbereich in dem keine oder nur wenige Übergaben für eine weitere Optimierung erzeugt werden. Wie zu erwarten war, bestimmt also die Kommunikationsrate die Geschwindigkeit der Optimierung.

5.4.4 Kommunikationspartner

Die Anzahl der Kommunikationspartner eines Knotens gibt Auskunft darüber, zu wie vielen anderen Knoten Nachrichten gesendet werden können. Der Simulator geht hierbei von einem unidirektionalen Nachrichtenaustausch aus. Das bedeutet, dass ein Knoten zwar eine Nachricht von einem anderen Knoten empfangen kann, aber an diesen Knoten möglicherweise keine Nachricht sendet.

In Tabelle 5.10 sind die Evaluationsergebnisse der vier Übergabestrategien bei einer unterschiedlichen Anzahl an Kommunikationspartnern aufgeführt. Die Er-

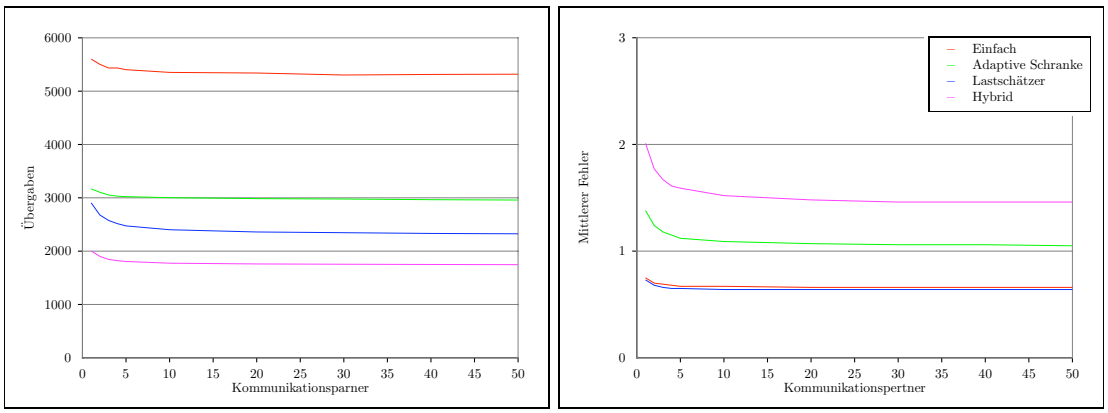

Abbildung 5.12: Anzahl der Übergaben und mittlerer Fehler in Abhängigkeit der Anzahl an Kommunikationspartner der Knoten

Tabelle 5.10: Einfluss der Kommunikationspartner

Strategie	Partner	$\bar{t}$	σ_t	$\overline{\delta_{nload_{\text{avg}}}}$	$\sigma_{\delta_{nload_{\text{avg}}}}$
Einfach	1	5601,65	101,68	0,75	0,025
	2	5502,10	100,45	0,70	0,022
	3	5433,85	95,24	0,69	0,018
	5	5402,36	96,54	0,67	0,018
	10	5351,26	93,50	0,67	0,019
	20	5339,61	86,50	0,66	0,021
	30	5302,52	85,31	0,66	0,020
	40	5312,87	98,43	0,66	0,019
	50	5317,58	87,61	0,66	0,022
Adaptive Schranke	1	3166,63	54,52	1,38	0,041
	2	3103,28	50,30	1,24	0,037
	3	3052,21	52,99	1,18	0,030
	5	3021,49	49,53	1,12	0,028
	10	3002,32	44,13	1,09	0,030
	20	2984,72	52,84	1,07	0,027
	30	2976,85	44,43	1,06	0,028
	40	2964,87	51,61	1,06	0,028
	50	2957,99	49,61	1,05	0,028
Lastschätzer	1	2904,98	66,08	0,73	0,027
	2	2677,09	52,67	0,68	0,018
	3	2575,32	42,28	0,66	0,020
	5	2473,01	40,82	0,65	0,020
	10	2401,25	39,89	0,64	0,017
	20	2359,21	38,10	0,64	0,021
	30	2345,80	38,08	0,64	0,020
	40	2331,74	35,74	0,64	0,018
	50	2325,01	32,35	0,64	0,020
Hybrid	1	2004,47	42,55	2,01	0,059
	2	1901,25	32,97	1,77	0,050
	3	1844,27	33,13	1,67	0,050
	5	1805,71	30,19	1,59	0,046
	10	1773,84	31,81	1,52	0,043
	20	1759,70	29,21	1,48	0,041
	30	1754,36	32,66	1,46	0,045
	40	1749,85	29,60	1,46	0,047
	50	1746,14	30,67	1,46	0,036

Kommunikationspartner und das rechte Diagramm den dabei erzielten mittleren Fehler.

Für alle Übergabestrategien kann festgestellt werden, dass die Anzahl der Übergaben sinkt und der mittlere Fehler kleiner wird, je mehr Kommunikationspartner verfügbar sind. Offensichtlich arbeiten die Übergabestrategien besser, wenn Informationen von mehreren unterschiedlichen Knoten gesammelt werden können.

Im Fall der Übergabestrategie mit Lastschätzer, mit adaptiver Schranke und dem Hybridverfahren ist dieses Verhalten leicht erklärbar. Mit mehr Information von unterschiedlichen Quellen ist eine genauere Einschätzung der vorhandenen Lasten und somit der mittleren Belastung der Knoten möglich. Dies führt zu einer besseren Anpassung der adaptiven Schranke aber auch der geschätzten mittleren Last des Netzwerks. Eine höhere Anzahl unterschiedlicher Kommunikationspartner erzielt also eine positive Wirkung auf den Selbstorganisationsprozess.

Bei einem Kommunikationspartner können alle vier Übergabestrategien nicht den Optimalwert erreichen. Mit jedem weiteren Kommunikationspartner nähern sich die Ergebnisse sehr schnell den jeweiligen Optimalwerten wie sie ab etwa 10 Kommunikationspartnern erreicht werden. Bei mehr als 20 Kommunikationspartnern ergeben sich nur noch marginale Verbesserungen.

In den betrachteten Simulationen wurden 1000 Knoten simuliert, was bedeutet, dass ab einem Prozent Vernetzung, also 10 Knoten, optimale Resultate erzielt werden können. Bei einer so hohen Anzahl von Knoten wird im realen System jedoch meist eine deutlich höhere Vernetzung der Knoten untereinander vorhanden sein.

Dies lässt den Schluss zu, dass das Selbstoptimierungsverfahren bereits für kleine Netze mit wenigen Knoten geeignet ist. Es ist sogar zu erwarten, dass bei kleinen Netzen mit einer vollständigen Vermaschung bessere Ergebnisse erzielt werden, da bereits nach wenigen Nachrichten bei allen Knoten ein vollständiges „Bild“ der vorhandenen Lasten im Netz entsteht und somit die Selbstoptimierung schneller den Optimalwert erreichen kann.

5.5 Dynamisches Prozessverhalten

Den bisherigen Betrachtungen der Übergabestrategien lag ein statisches Verhalten der zu betrachtenden Dienste zugrunde. Ausgehend von dem initialen Ressourcenverbrauch der einzelnen Dienste wurde die Verteilung der Dienste so optimiert, dass bei unveränderten Belastungen eine möglichst gleichmäßige Belastung der Knoten gewährleistet wird.

Die Annahme, dass sich Dienste in ihrem Ressourcenverbrauch nicht oder nur kaum ändern, trifft nur für Systeme zu, deren Belastungszustände im Betrieb

weitestgehend unverändert bleiben oder wenn die ermittelten Werte der Dienste über sehr lange Zeiträume betrachtet werden. In beiden Fällen kann für einen bestimmten Zeitraum angenommen werden, dass der Ressourcenverbrauch der einzelnen Dienste unverändert bleibt.

In Systemen, bei denen häufige Lastwechsel und stark unterschiedliche Belastungen der Dienste zu beobachten sind, muss der Zeitraum für die Berechnung des Ressourcenverbrauchs verkürzt werden, um den realen Gegebenheiten Rechnung zu tragen. Für diese Situation muss von einem dynamischen Verhalten der Dienste ausgegangen werden.

Da sich ändernde Ressourcenbelastungen zu ständig neuen Verhältnissen innerhalb des Netzwerks führen, müssen die Übergabestrategien auf dieses Verhalten hin untersucht werden. Wünschenswert wäre eine Adaption der Übergabestrategien an wechselnde Belastungsverhältnisse, damit eine Oszillation bei der Verlegung der Dienste vermieden wird.

Eine Oszillation kann beispielsweise auftreten, wenn ein Dienst auf einem Knoten kurzzeitig einen größeren Ressourcenbedarf erfährt, auf einen anderen Knoten verlegt wird und nach Normalisierung des Ressourcenbedarfs wieder auf den Ursprungsknoten zurückverlegt wird oder von Knoten zu Knoten wandert.

Nachfolgend sollen zwei typische Arten dynamischen Prozessverhaltens betrachtet werden. Lastspitzen, die sich zyklisch auf eine Anzahl von Diensten Auswirkung und deren Ressourcenbedarf zeitweise verändert, sowie eine kontinuierliche Lastzunahme, die über einen gewissen Zeitraum erhalten bleibt und anschließend wieder auf das Ausgangsniveau zurückkehrt.

5.5.1 Lastspitzen

Lastspitzen stellen Änderungen im Ressourcenbedarf dar, die sich wiederkehrend auf einen oder mehrere Knoten auswirken und den Ressourcenbedarf sowohl positiv wie auch negativ für eine gewisse Zeit verändern. Abbildung 5.13 zeigt eine schematische Darstellung positiver Lastspitzen.

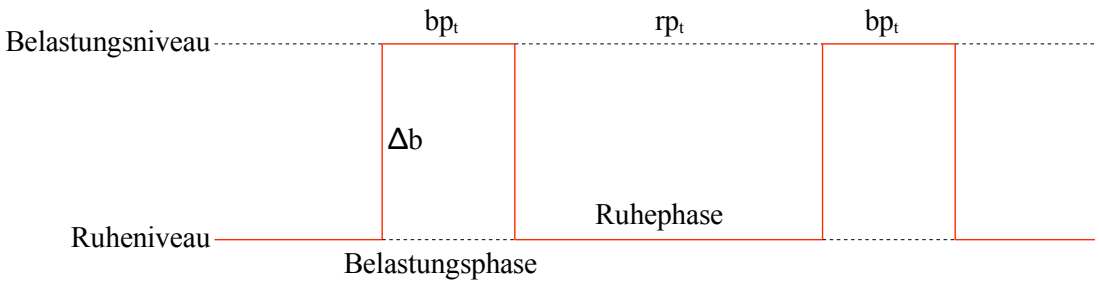

Abbildung 5.13: Definition der Parameter von Lastspitzen in der Simulation

Ausgehend vom normalen Ressourcenverbrauch des Dienstes (Ruheniveau) verändert sich der Ressourcenverbrauch um Δb auf das Belastungsniveau. Das Belastungsniveau bleibt für den Zeitraum von bp_t Simulationsschritten erhalten. Anschließend kehrt der Ressourcenverbrauch wieder auf den Wert des Ruheniveaus zurück und verweilt dort für die Zeit von rp_t. Darauf folgt wieder eine Lastspitze mit dem bereits beschriebenen Verhalten.

Das Verhältnis zwischen der Belastungs- und der Ruhephase, sowie die Größe der Laständerung ist ein wichtiger Faktor für die Dynamik der Dienste. Für einen Dienst s ergibt sich damit eine durchschnittliche Laständerung von

$$\Delta B_s = \Delta b * \frac{bp_t}{bp_t + rp_t} \tag{5.26}$$

wobei Δb die prozentuale Veränderung der Ressource angibt. Bei einem Verhältnis von $bp_t = rp_t$ und $\Delta b = 50\%$ erfährt der betroffene Dienst im Schnitt eine Laständerung von 25 Prozent.

In der Simulation kann außer den bereits besprochenen Parametern, zusätzlich die prozentuale Anzahl der Dienste Δs definiert werden, die Lastspitzen erfahren sollen. Somit ist die Laständerung im gesamten Netz

$$\Delta B = \Delta s * \Delta b * \frac{bp_t}{bp_t + rp_t} \tag{5.27}$$

Variation des Belastungsverhältnisses

Das Belastungsverhältnis gibt das Verhältnis zwischen Belastungs- und Ruhephasen der Dienste an. Die folgenden Simulationen wurden mit 25 Prozent dynamischer Dienste durchgeführt, deren Last sich um maximal 25 Prozent ihrer Grundlast ändert.

Die Diagramme in Abbildung 5.14 zeigen die Evaluationsergebnisse für Belastungsverhältnisse von 40:60, 60:40 und 80:20. Dabei gibt der erste Wert die Belastungsdauer bp_t und der zweite die Ruhedauer rp_t an. Als Intervall wurden 100 Simulationsschritte gewählt. Das bedeutet, dass bei einem Lastverhältnis von 80:20 die betroffenen Dienste 80 Simulationsschritte lang eine erhöhte Last erfahren und 20 Simulationsschritte auf dem normalen Belastungsniveau verweilen.

Auf der linken Seite ist jeweils die Anzahl der Übergaben der einzelnen Strategien dargestellt und auf der rechten Seite die zugehörigen mittleren Fehler. Die einfache Übergabestrategie und die Übergabestrategie mit Lastschätzer sind nicht in der Lage, die Dynamik des Systems zu erkennen und versuchen folglich jede

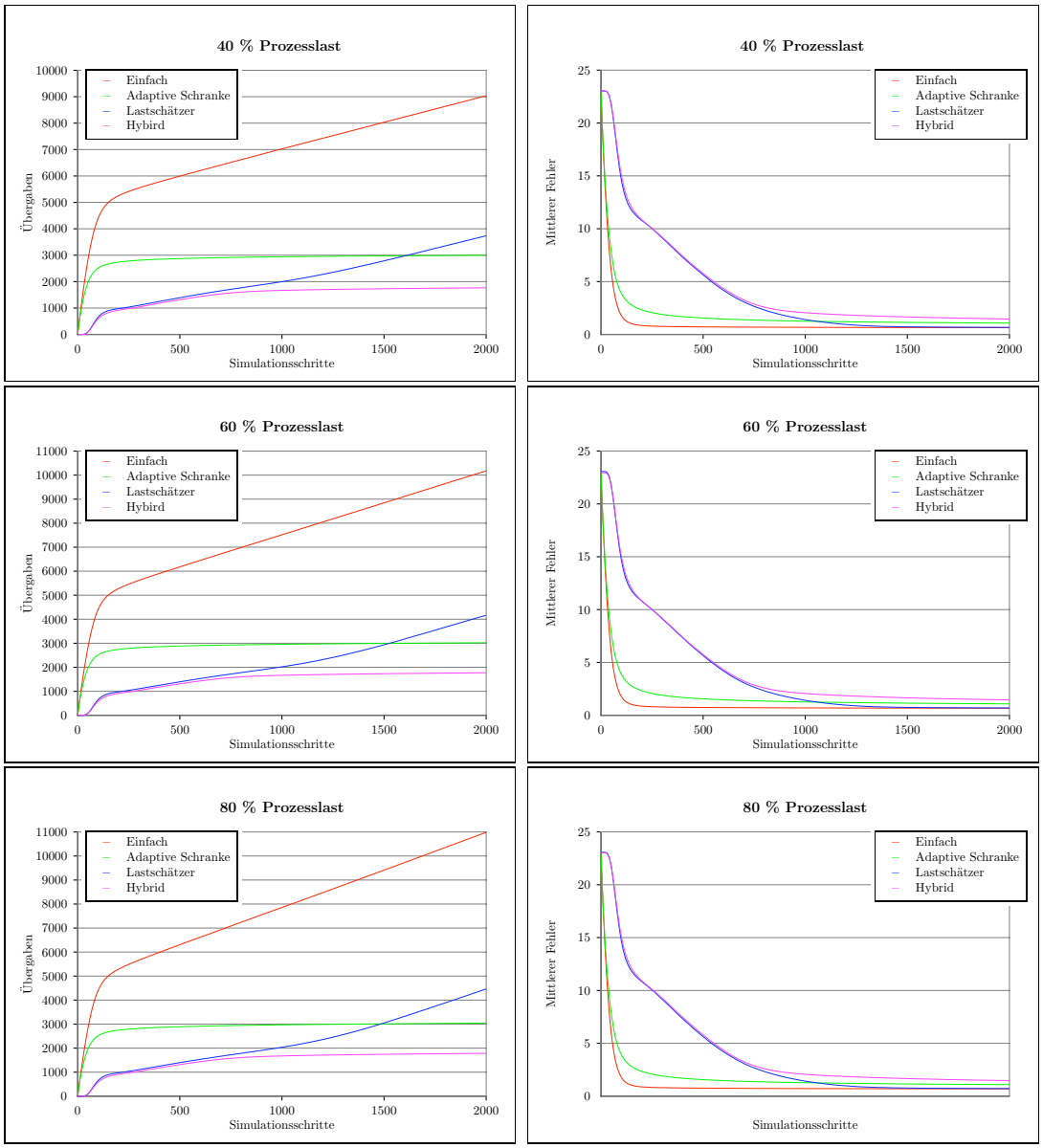

Abbildung 5.14: Anzahl der Übergaben und mittlerer Fehler bei Lastverhältnissen von 40, 60 und 80 Prozent

mögliche Optimierung durchzuführen. Dies führt zu einem Anstieg der Übergaben bei andauernder Dynamik der Dienste.

Bei den Übergabestrategien mit adaptiver Schranke und Hybridverfahren lernt die adaptive Schranke die Dynamik im System und kann dadurch unnötige Verlegungen von Diensten unterbinden. Selbst bei einem Lastverhältnis von 80:20 ist die adaptive Schranke in der Lage, den mittleren Gewinn für eine Verlegung so

gut zu bestimmen, dass das dynamische Verhalten der Prozesse nicht als Optimierungspotential betrachtet wird.

Der mittlere Fehler verhält sich bei allen Simulationen annähernd gleich. Unabhängig von dem gewählten Belastungsverhältnis sind alle Übergabestrategien in der Lage, die aus dem statischen Fall bekannten mittleren Fehler zu erreichen. Dies lässt den Schluss zu, dass die Art der Nachrichtenübertragung, obwohl sie zufällig geschieht, durchaus sehr robust ist und zuverlässig die Informationen an alle Knoten verteilt.

Variation der relativen Laständerung

Die relative Laständerung entspricht dem prozentualen Anteil der Grundlast, der während der Belastungsphase bp_t eines Dienstes hinzugefügt oder abgezogen wird. Eine Laständerung um 50 Prozent bedeutet, dass zu der Grundlast des Dienstes bis zu 50 Prozent der Grundlast hinzu kommt oder davon abgezogen wird. Damit kann der Dienst 150 Prozent oder 50 Prozent seiner normalen Last während der Belastungsphase aufweisen.

In Abbildung 5.15 sind die Ergebnisse der Simulationen bei Laständerungen von 25, 50, 75 und 100 Prozent dargestellt. Rechts daneben ist der mittlere Fehler der einzelnen Übergabestrategien abgebildet. Für die Simulationen wurde angenommen, dass sich 25 Prozent der Dienste dynamisch verhalten und dass für diese Dienste ein Belastungsverhältnis von 20:80 besteht.

Die Diagramme zeigen für die einfache Übergabestrategie und die Übergabestrategie mit Lastschätzer einen enormen Anstieg in der Anzahl der Übergaben. Bei einer vier mal so großen Laständerung werden etwa vier mal so viele Dienste verlegt. Für die einfache Übergabestrategie wurden bei 25 Prozent Laständerung nach 2000 Simulationsschritten etwas weniger als 8000 Übergaben benötigt, bei 100 Prozent Laständerung fast 32000. Das gleiche Verhalten zeigt auch die Übergabestrategie mit Lastschätzer, jedoch mit einer deutlich geringeren Anzahl an Übergaben.

Dieses Verhalten ist damit zu erklären, dass nicht die Dienste verlegt werden, die ihre Last verändern, sondern die Dienste mit kleiner Basislast, die zum Ausgleich der veränderten Verhältnisse auf einen anderen Knoten verlegt werden.

Die adaptive Schranke der beiden anderen Verfahren ist auch für den Fall der Laständerung in der Lage, fast alle Übergaben, die durch die Laständerung auftreten zu unterdrücken. Die Schranke passt ihre Größe aufgrund des ständig vorhandenen Optimierungspotentials so weit nach oben an, dass nur die Dienste verlegt werden, deren Laständerung oberhalb des berechneten Gewinns liegt. Wie Abbildung 5.16 zeigt, führt dies dazu, dass Laständerungen bis 75 Prozent nur einen geringen Anstieg in der Anzahl der Übergaben hervorrufen.

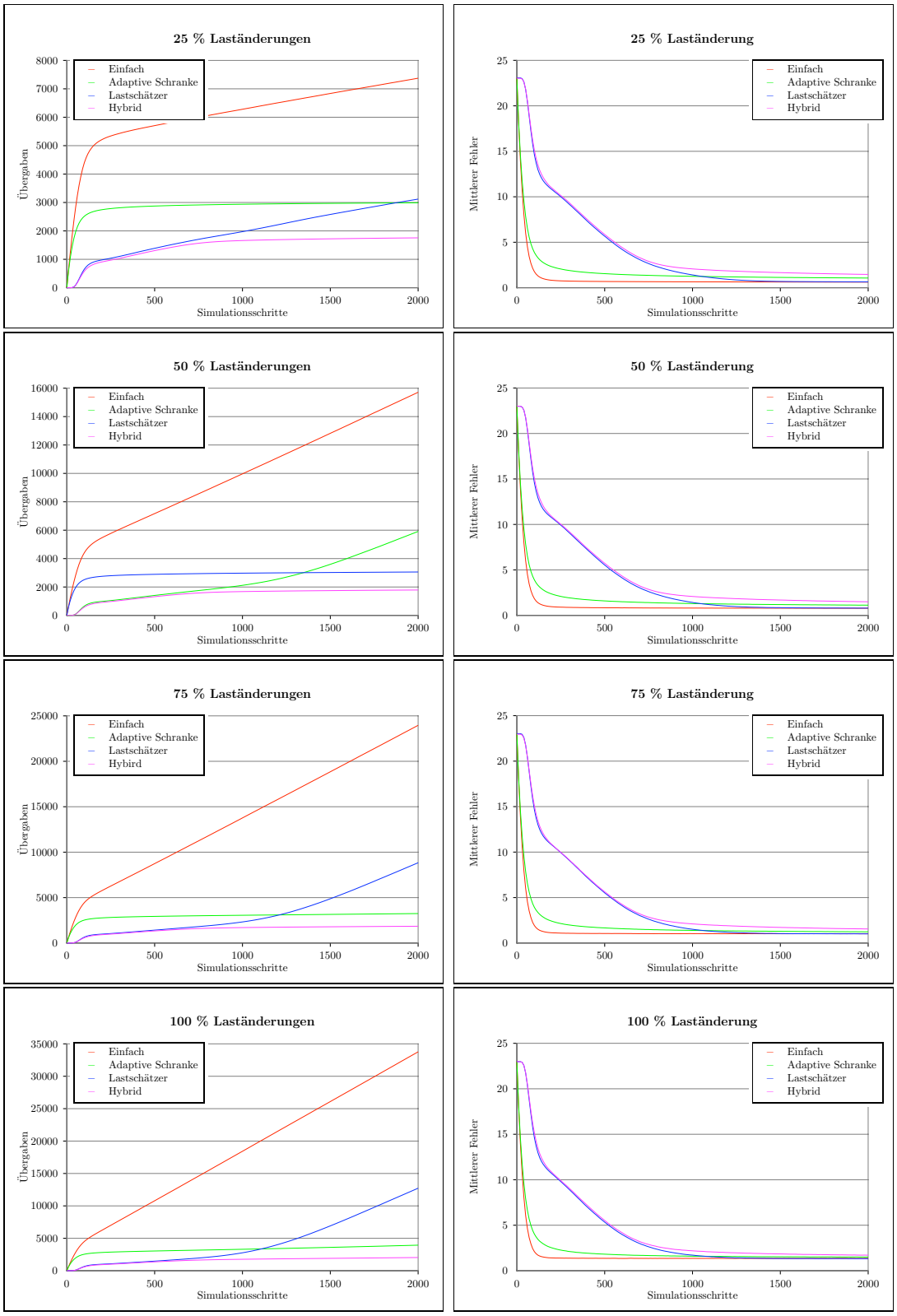

Abbildung 5.15: Anzahl der Übergaben und mittlerer Fehler bei Laständerungen um 25, 50, 75 und 100 Prozent der Dienstlast

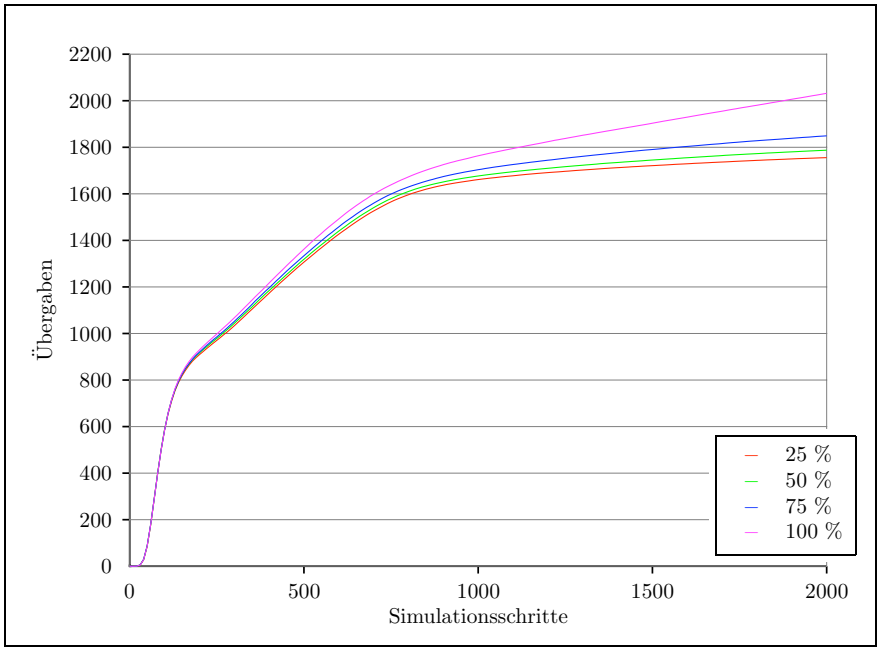

Abbildung 5.16: Anzahl der Übergaben der Strategie mit Hybridverfahren bei Laständerungen um 25, 50, 75 und 100 Prozent der Dienstlast

Bemerkenswert ist auch, dass bei allen Simulationen sehr kleine Werte für den mittleren Fehler erreicht werden. Obwohl die Prozesse ihre Last lediglich für 20 Simulationsschritte beibehalten, ist dies für die Strategien ausreichend, um auf die veränderten Lastverhältnisse angemessen zu reagieren und nachhaltig die Last im System gleichmäßig zu verteilen.

Variation des prozentualen Anteils belasteter Dienste

Ein weiterer Parameter zur Variation der Laständerung ist der prozentuale Anteil dynamischer Dienste. Bei einem prozentualen Anteil von 50 Prozent weist die Hälfte der Dienste ein dynamisches Verhalten auf.

Die Simulationen wurden mit einem Belastungsverhältnis von 20:80 und einer maximalen Laständerung von 25 Prozent durchgeführt. Die Simulationsergebnisse sind in Abbildung 5.17 dargestellt. Links ist die Anzahl der Übergaben und rechts der zugehörige mittlere Fehler der einzelnen Übergabestrategien abgebildet.

Wie bereits bei der Variation der Belastungsänderung, kann für den Fall der einfachen Übergabestrategie und der Übergabestrategie mit Lastschätzer auch hier ein proportionales Verhalten zwischen der Anzahl der belasteten Dienste und der Anzahl der Übergaben beobachtet werden. Bei 100 Prozent dynamischer Dienste werden vier mal so viele Übergaben benötigt, wie bei 25 Prozent.

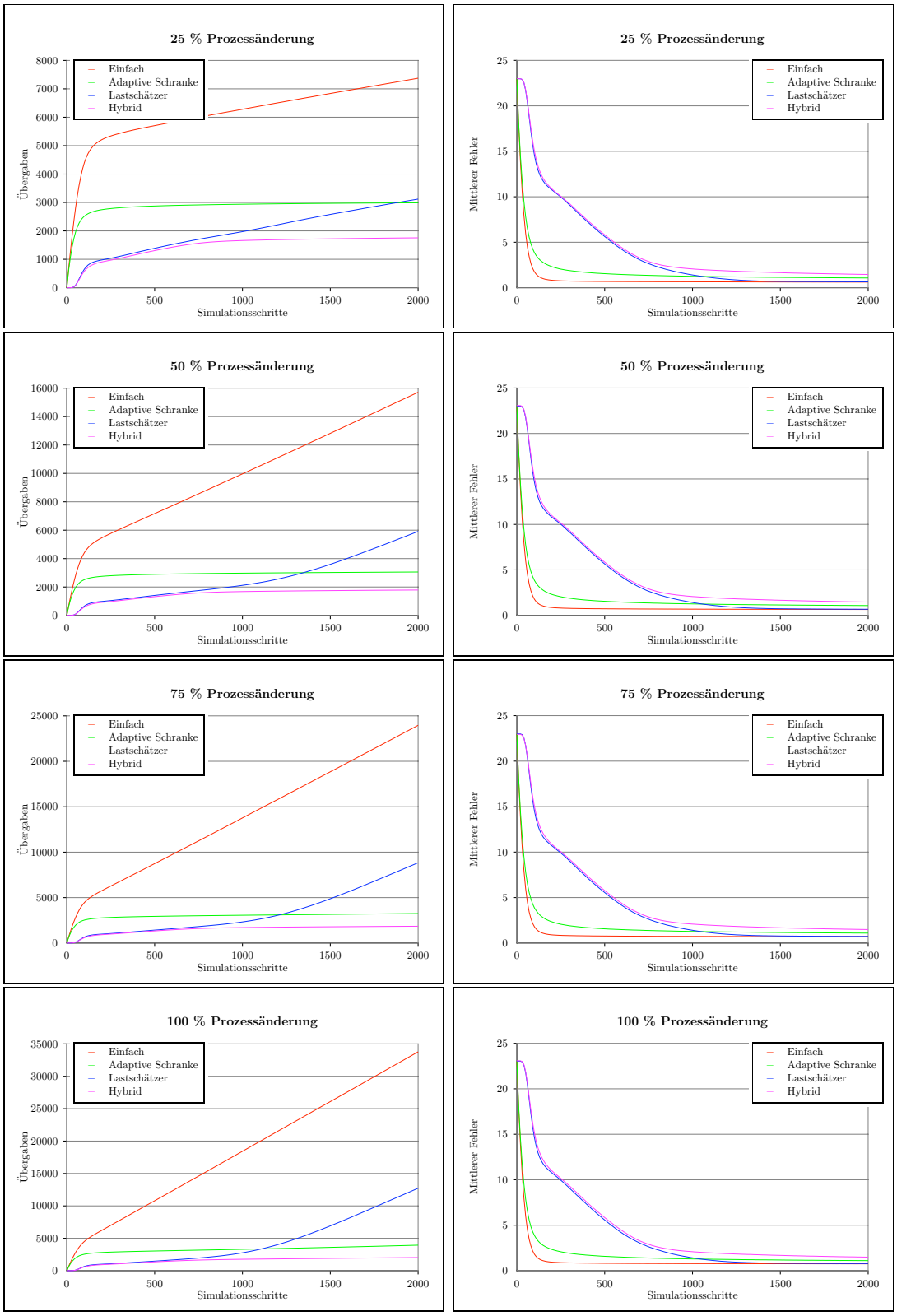

Abbildung 5.17: Anzahl der Übergaben und mittlerer Fehler bei 25, 50, 75 und 100 Prozent dynamischer Dienste

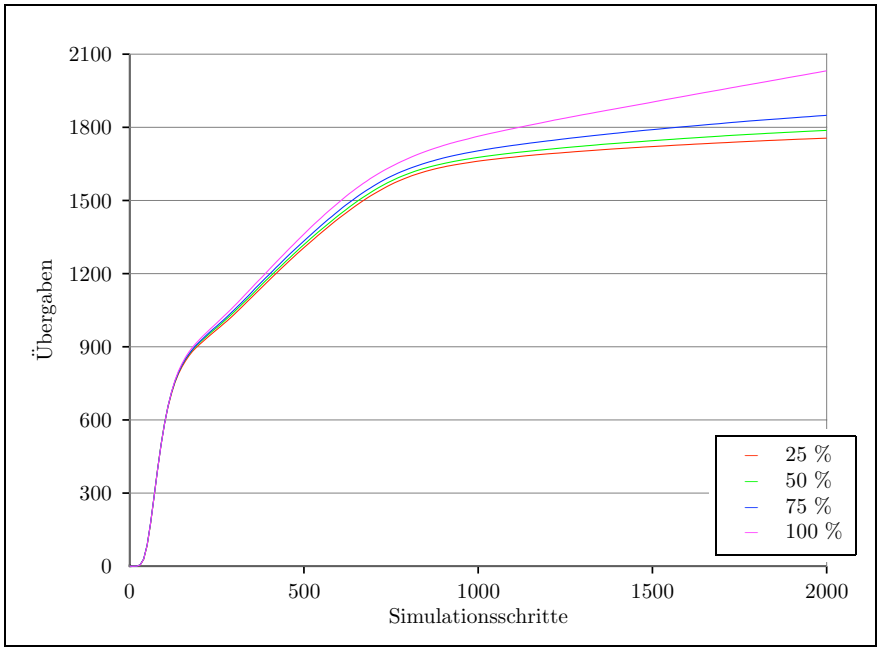

Abbildung 5.18: Anzahl der Übergaben der Strategie mit Hybridverfahren bei 25, 50, 75 und 100 Prozent dynamischer Dienste

Offensichtlich scheint es für die Übergabestrategie nicht von Bedeutung zu sein, wie die Laständerung zustande kommt, da für beide Arten der Laständerung, bei gleichen Simulationsbedingungen, die gleiche Größenordnung an Übergaben erzeugt wird. Dies stützt die These, dass nicht die Dienste verlegt werden, die eine Laständerung erfuhren, sondern die Dienste, die zum Ausgleich der entstandenen Unterschiede herangezogen werden.

Damit ist zu erwarten, dass für die beiden Übergabestrategien mit adaptiver Schranke ein ähnliches Ergebnis wie zuvor entsteht. Die Evaluationsergebnisse bestätigen, auch für den Fall einer Änderung der Anzahl der dynamischen Dienste, dass die adaptive Schranke in der Lage ist, Lastschwankungen im System zu lernen und sich ändernden Lastsituationen anzupassen.

Das gute Adaptionsvermögen der Übergabestrategie mit Hybridverfahren wird in Abbildung 5.18 durch die geringe Anzahl zusätzlicher Übergaben bestätigt. Es zeigt sich das gleiche Verhalten wie in den Simulationen zuvor. Erst bei einem prozentualen Anteil von über 75 Prozent dynamischer Dienste steigt die Anzahl der Übergaben leicht an.

Der mittlere Fehler verhält sich auch hier wie in den vorherigen Simulationen. Es wird für alle Strategien ein kleiner mittlerer Fehler erzielt, der für die Güte der Selbstoptimierung ausschlaggebend ist. Selbst bei 100 Prozent dynamischer Dienste ist die Selbstoptimierung in der Lage einen kleinen mittleren Fehler zu erzielen,

wobei die notwendige Anzahl verlegter Dienste zwischen den Übergabestrategien stark schwankt.

5.5.2 Kontinuierliche Lastzunahme

Eine kontinuierliche Lastzunahme verläuft, im Gegensatz zu den bereits betrachteten periodischen Lastspitzen, in einem längeren Zeitraum und führt mittels einer kontinuierlich steigenden Last über einen bestimmten Zeitraum zum Belastungsniveau. Am Ende der Belastungsphase wird die erzeugte Belastung wieder über einen Zeitraum zum Ruheniveau hin abgebaut.

Bei der kontinuierlichen Lastzunahme wird davon ausgegangen, dass alle Knoten gleichzeitig eine Laständerung erfahren. Im Gegensatz zu den Lastspitzen, bei denen Dienste zu unterschiedlichen Zeitpunkten eine Laständerung erfahren, sind hier alle Dienste gleichzeitig betroffen. Wurde zuvor von den Übergabestrategien gefordert, dass Lastspitzen toleriert werden sollen, um unnötige Dienstverlegungen zu verhindern, sollen die Simulationen der kontinuierlichen Lastzunahme zeigen, dass die Übergabestrategien in der Lage sind, einen Belastungsanstieg, den das ganze System betrifft, zu erkennen und mit möglichst wenig Übergaben auszugleichen.

Abbildung 5.19 zeigt schematisch den Verlauf einer kontinuierlichen Lastzunahme. Über einen Zeitraum von b_t Simulationsschritten wird das Belastungsniveau erreicht, das um Δb über dem Ruheniveau liegt. In diesem Zeitraum wird bei den betroffenen Diensten der Ressourcenbedarf schrittweise bis zum Belastungsniveau erhöht.

Die Belastung bleibt, wie bereits bei den Lastspitzen auch, über einen Zeitraum von bp_t Simulationsschritten bestehen. Anschließend wird der Ressourcenverbrauch der Dienste wieder schrittweise auf das Ruheniveau gesenkt.

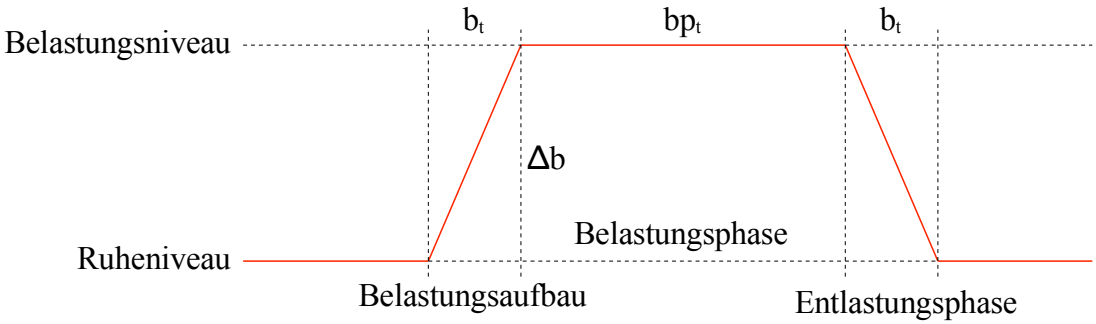

Abbildung 5.19: Definition der Parameter einer kontinuierlichen Laständerung in der Simulation

Wie bei den Lastspitzen, kann auch für die kontinuierliche Lastzunahme die Anzahl der Dienste definiert werden, die von der Laständerung betroffen sind. Aus der prozentualen Angabe der betroffenen Dienste wird in der Simulation die tatsächliche Anzahl der Dienste berechnet, die eine Laständerung erfahren sollen. Sei S die Anzahl aller Dienste im Netzwerk und s die Anzahl der von der Lastzunahme betroffenen Dienste, dann berechnet sich die Änderung der Gesamtlast im System zu

$$\Delta B = \frac{s}{S} * \Delta b \tag{5.28}$$

Variation der Belastungszeiten

Die Variation der Belastungszeiten soll verdeutlichen, wie die Übergabestrategien mit unterschiedlich langen Belastungsphasen umgehen. Wichtig ist dabei, dass die Übergabestrategien erkennen, dass die Gesamtlast im System steigt, somit die Notwendigkeit einer Optimierung besteht und diese mit möglichst wenig Übergaben erfolgt.

Die Simulationen wurden mit den Standardparametern der bekannten Simulationen durchgeführt. Zusätzlich wurde angenommen, dass 25 Prozent der Dienste eine Laständerung erfahren und dass die Laständerung $\Delta b = 5$ beträgt. Dieser Wert entspricht bei den Standardeinstellungen des Simulators der mittleren Last der Dienste und erzeugt dadurch im Mittel eine Laständerung im 100 Prozent bei den betroffenen Diensten.

In Abbildung 5.20 sind die Simulationsergebnisse für verschiedene Belastungszeiten und die zugehörigen mittleren Fehler abgebildet. Der Wert der Laständerung wird über den Zeitraum des Belastungsanstiegs b_t gleichmäßig verteilt. Für den Fall von $b_t = 5$ bedeutet dies, dass in jedem Simulationsschritt die Last der dynamischen Dienste um den Wert 1 erhöht wird. Für $b_t = 100$ wird somit die Last der dynamischen Dienste nur alle 20 Simulationsschritte um den Wert 1 erhöht.

Die Abbildungen der mittleren Fehler zeigen deutlich die Belastungsänderungen. Dabei ist zu erkennen, dass der mittlere Fehler größer ist, je schneller sich die Belastung ändert. Dies liegt daran, dass bei einer längeren Zeitdauer für den Belastungsanstieg b_t bereits Dienste verlegt werden, die den entstandenen Lastunterschied ausgleichen und dadurch den mittleren Fehler verringern.

Bei der Simulation mit $b_t = 5$ und $bp_t = 50$ erfolgt die Laständerung so schnell, dass die Übergabestrategien nicht in der Lage sind schnell genug die Dienste zu verlegen, um zwischen dem Anstieg und dem Abfallen der Belastung ein Gleichgewicht im System herzustellen. Wichtig ist jedoch die Erkenntnis, dass alle Strategien auf die Laständerung im gesamten Netz sehr schnell reagieren. Die

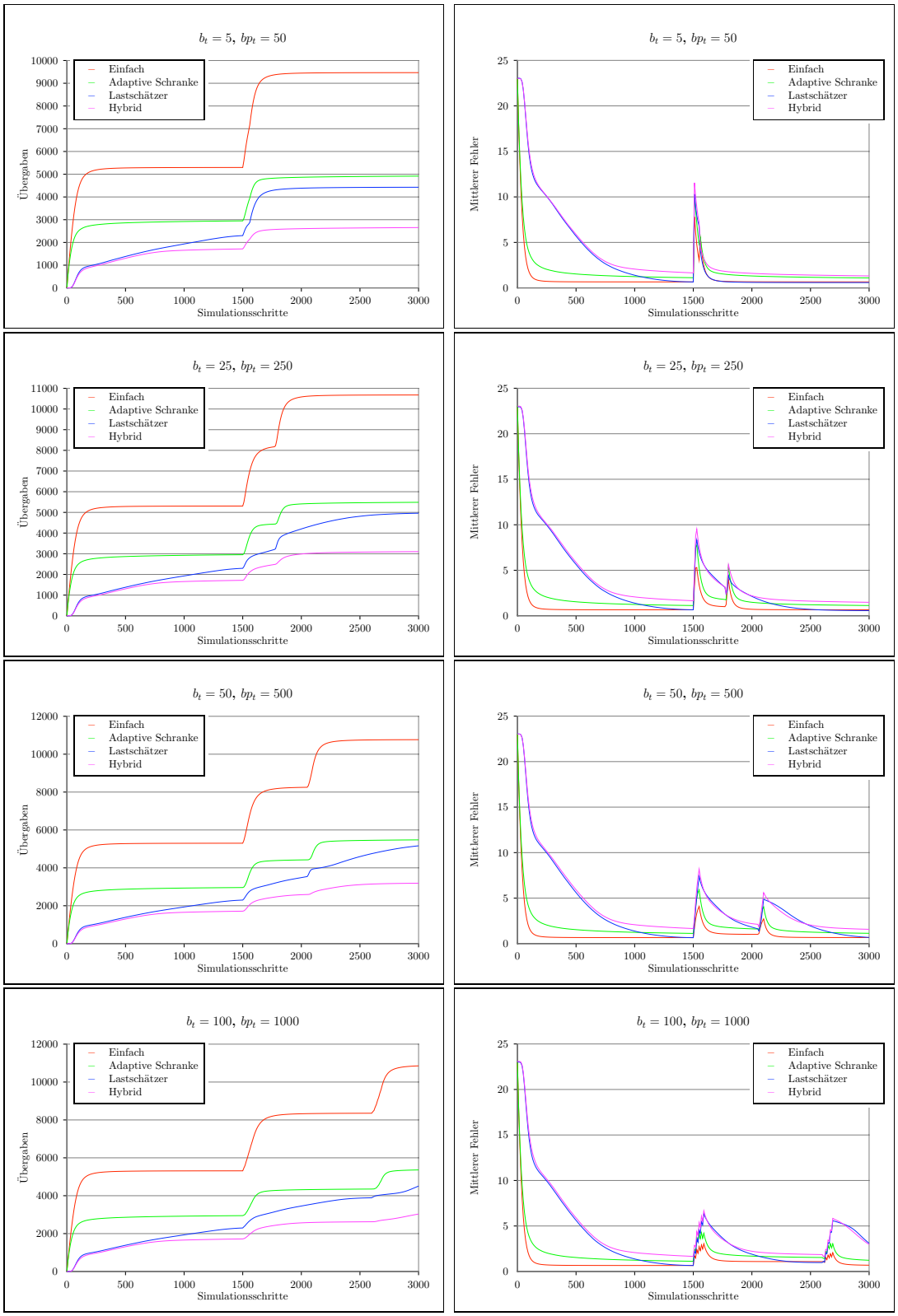

Abbildung 5.20: Anzahl der Übergaben und mittlerer Fehler bei unterschiedlichen Belastungszeiten

Stärke der Reaktion und damit die Anzahl der Übergaben sind wiederum zwischen den verschiedenen Übergabestrategien sehr unterschiedlich. Die einfache Übergabestrategie erzeugt sehr viele Übergaben, während die Übergabestrategie mit Hybridverfahren versucht einen Teil der Übergaben zu unterdrücken und damit insgesamt sehr viel weniger Übergaben benötigt. Der mittlere Fehler kehrt bei allen Strategien auf den Wert vor der Laständerung zurück.

Bei den weiteren Simulationen sind die Zeitpunkte der Lastzunahme und der Entlastung deutlich in den Diagrammen zu erkennen. Mit zunehmender Dauer der Plateauphase entfernen sich diese Zeitpunkte voneinander. Die Anzahl der Übergaben steigt bei keiner der Übergabestrategien durch Verlängerung der Belastungszeiten merklich an. Jedoch kann mit zunehmender Dauer der Plateauphase die Gesamtzahl der Übergaben in zwei gleich große Anteile unterteilt werden.

Dieses Verhalten ist dadurch zu erklären, dass die Strategien bei einer längeren Plateauphase in der Lage sind die Last zwischen den Knoten auszugleichen. Damit entsteht bei der Entlastung der Dienste wieder eine Veränderung der Gesamtlast, die in ihrer Größe der vorherigen Belastung entspricht und dadurch das gleiche Verhalten hervorruft.

An der Größe des mittleren Fehlers ist zu erkennen, dass die Übergabestrategien die neue Gesamtlast des Systems erkennen. Durch die größere Last der Dienste ist die Abweichung vom rechnerischen Optimum größer wodurch sich ein höherer Wert für den mittleren Fehler ergibt. Für die Werte $b_t = 50$ und $bp_t = 500$ ist dieses Verhalten deutlich zu erkennen.

Bei den Einstellungen $b_t = 100$ und $bp_t = 1000$ reichen die 3000 Simulationsschritte nicht mehr aus, damit die Strategien mit Lastschätzer das Netzwerk vollständig optimieren kann. Würde die Simulation länger laufen, würden auch diese beiden Strategien wie in der Simulation mit den Werte $b_t = 50$ und $bp_t = 500$ ihren Endwert erreichen.

Die Simulationen zeigen, dass alle Übergabestrategien in der Lage sind, eine Anstieg der Gesamtlast im System rasch zu erkennen und darauf zu reagieren. Die Anzahl der notwendigen Übergaben und die Geschwindigkeit der Optimierung sind von der gewählten Strategie abhängig. Auch bei den dargestellten Simulationen bewährt sich die Übergabestrategie mit Hybridverfahren durch eine angemessene Reaktionszeit bei minimaler Anzahl notwendiger Übergaben.

Variation der Anzahl belasteter Dienste

Um die Auswirkungen einer globalen Belastungsänderung auf das System zu untersuchen, wurde für die folgenden Simulationen die Anzahl der Dienste, die von einer Laständerung betroffen sind, variiert. Die Werte der Belastungszeiten wurden mit $b_t = 50$ und $bp_t = 500$ so gewählt, dass die Belastungsphase und Entla-

stungsphase voneinander getrennt sind, damit die Übergabestrategien das System optimiert haben, bevor die Entlastungsphase eintritt.

Die Größe der Lastzunahme wurde mit $\Delta b = 5$ wieder so gewählt, dass sie bei den Standardeinstellungen des Simulators im Mittel eine Laständerung der Dienste um 100 Prozent bewirkt.

Abbildung 5.21 zeigt die Evaluationsergebnisse bei 25, 50, 75 und 100 Prozent Anteil dynamischer Dienste. Die linken Diagramme zeigen die Anzahl der Übergaben und die rechten Diagramme den mittleren Fehler der einzelnen Übergabestrategien.

Das erste Diagramm mit 25 Prozent dynamischer Prozesse ist bereits aus der vorherigen Simulation bekannt und dient gewissermaßen als Referenz für die übrigen Simulationen. Bei einem Anteil von 50 Prozent dynamischer Prozesse ist ein deutlicher Anstieg des mittleren Fehlers bei den Übergabestrategien mit Lastschätzer zu erkennen. Dieser Anstieg liegt darin begründet, dass die Übergabestrategien mit Lastschätzer nicht so schnell in der Lage sind Dienste zu verlegen, da immer passende Partner für eine Übergabe gefunden werden müssen. Da die Kommunikationspartner zufällig ausgewählt werden, dauert es eine gewisse Zeit, bis Knoten gefunden werden, die in der Lage sind einen Dienst auszutauschen, um die Gesamtlast des Systems und damit den mittleren Fehler zu reduzieren. Die Gesamtzahl der Übergaben bleibt jedoch unverändert.

Ein zunächst unerwartetes Verhalten aller Übergabestrategien tritt ab 75 Prozent dynamischer Dienste ein. Aus dem Diagramm der Übergaben ist zu erkennen, dass deutlich weniger Übergaben zum Ausgleich der Last benötigt werden. Der mittlere Fehler steigt auch nicht weiter an, sondern wird bei den Übergabestrategien mit Lastschätzer lediglich etwas langsamer abgebaut.

Dieses Verhalten der Übergabestrategien ist dadurch zu erklären, dass sich ab einem Anteil von über 50 Prozent dynamischer Dienste die Verhältnisse im System umkehren. Da bei 75 Prozent drei Viertel der Dienste eine Laständerung erfahren verbleibt lediglich ein Viertel der Dienste auf ihrem alten Belastungsniveau. Somit stehen weniger Dienste mit geringer Last zum Ausgleich der Lastunterschiede zwischen den Knoten zur Verfügung und die Übergabestrategien mit Lastschätzer und mit Hybridverfahren benötigen mehr Zeit, um die entsprechenden Dienste für den Lastausgleich zu finden. Insgesamt werden dadurch weniger Übergaben benötigt.

Dieser Trend verstärkt sich weiter, wenn alle Dienste eine Lastzunahme erfahren. Die Anzahl der Übergaben sinkt weiter deutlich ab. Bei der Übergabestrategie mit Hybridverfahren führt dies dazu, dass nahezu keine Übergaben erfolgen.

Anhand des Diagramms der mittleren Fehlern kann das Verhalten der Strategien erklärt werden. Da alle Dienste nahezu gleichzeitig eine Laständerung erfahren, steigt der mittlere Fehler weniger stark an wie zuvor.

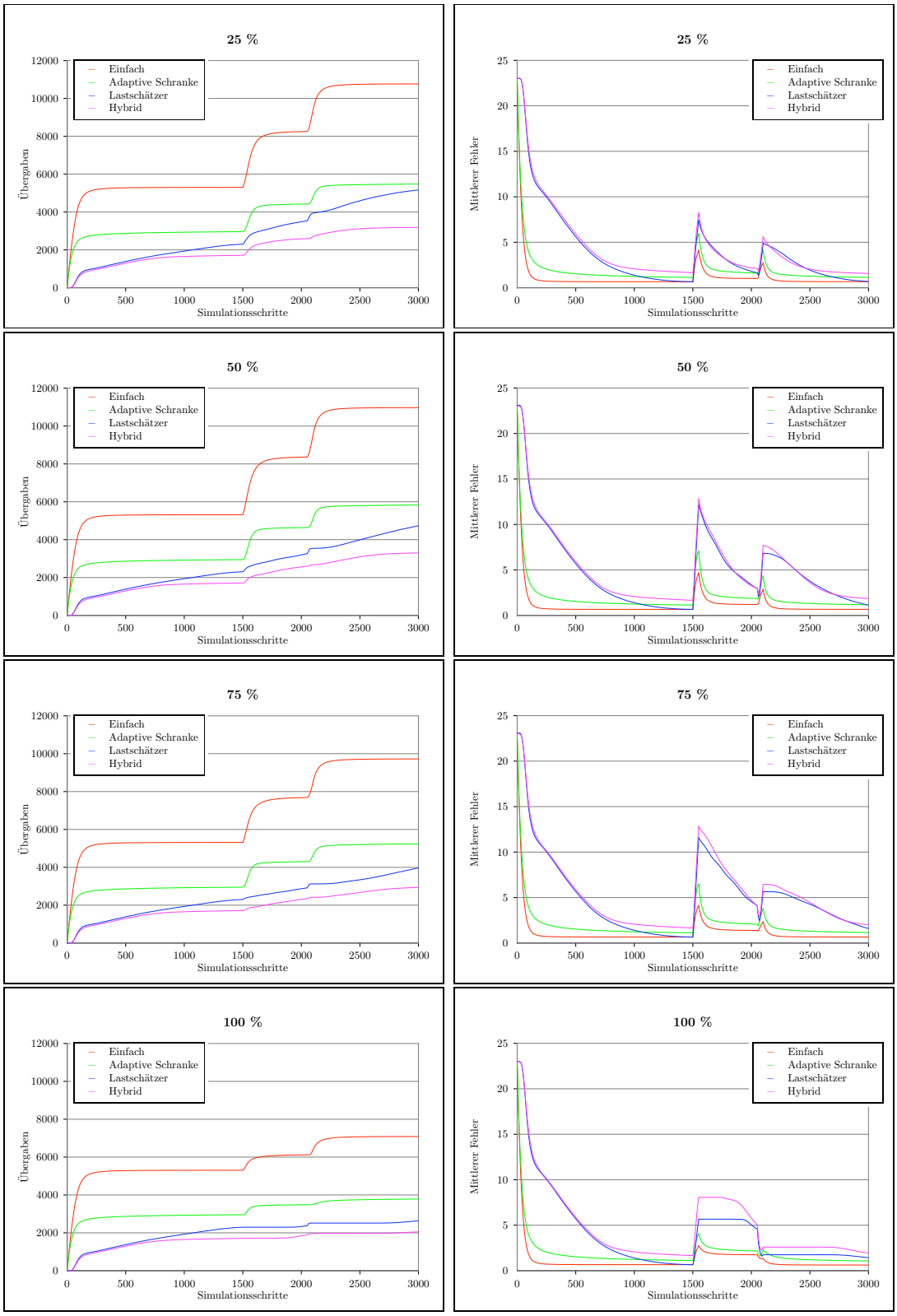

Abbildung 5.21: Anzahl der Übergaben und mittlerer Fehler bei 25, 50, 75 und 100 Prozent Anteil dynamischer Prozesse

Die Übergabestrategien mit Lastschätzer sind darüber hinaus nicht mehr in der Lage passende Knoten für eine Übergabe zu finden. Aus diesem Grund verweilt der mittlere Fehler bei diesen Strategien zunächst auf einem relativ hohen Niveau während der Plateauphase bevor er beginnt langsam zu sinken. Mit einsetzen der Entlastungsphase fällt der mittlere Fehler schnell auf einen Wert nahe dem Ausgangswert zurück.

Das bedeutet, dass der Lastschätzer erkennt, dass alle Knoten eine Lasterhöhung erfahren haben, und dass es keinen Sinn macht das System zu optimieren. Aus diesem Grund steigt auch der mittlere Fehler um etwa den Betrag der Lastzunahme ($\Delta b = 5$) an, da sich alle Dienste sich um diesen Betrag vom rechnerischen Optimum entfernen. Wird die Last aus dem System entfernt, reduziert sich auch der mittlere Fehler wieder um denselben Betrag.

Die Übergabestrategie mit Hybridverfahren zeigt also auch in Systemen mit gleichmäßigen und starken Laständerungen hervorragende Ergebnisse, da die Laständerungen erkannt werden und der neue Zustand des Systems adaptiert wird. Die adaptive Schranke verzögert dabei die Übergabe von Diensten lange genug, so dass der Lastschätzer die neuen Verhältnisse richtig einschätzen kann.

Würde die Plateauphase bei der Simulation mit 100 Prozent dynamischer Dienste länger andauern, würde die Übergabestrategie mit Hybridverfahren durch Ausgleich der Lasten den mittleren Fehler reduzieren und wie bisher einen mittleren Fehler etwas oberhalb der anderen Strategien erzielen.

5.6 Integration in die Middleware

Für die Integration der Selbstorganisation in die Middleware konnten lediglich die Algorithmen als Vorlage benutzt werden. Die Klassen mussten neu implementiert werden, da der Simulator auf die Simulation großer Netze ausgelegt und dementsprechend im Aufbau der Klassen für diesen Fall optimiert ist.

5.6.1 Komponenten der Selbstoptimierung

Die Selbstoptimierung der Middleware besteht aus mehreren Komponenten, die in verschiedenen Bereichen der Middleware angesiedelt sind. Abbildung 5.22 zeigt die wichtigsten Komponenten, die zur Selbstoptimierung benötigt werden.

Der `RelocationHandlingService` übernimmt die Steuerung der Selbstoptimierung. Er veranlasst die Verlegung eines Dienstes auf einen anderen Knoten, sowie das Starten eines Dienstes, der von einem anderen Knoten auf den lokalen Knoten verlegt werden soll.

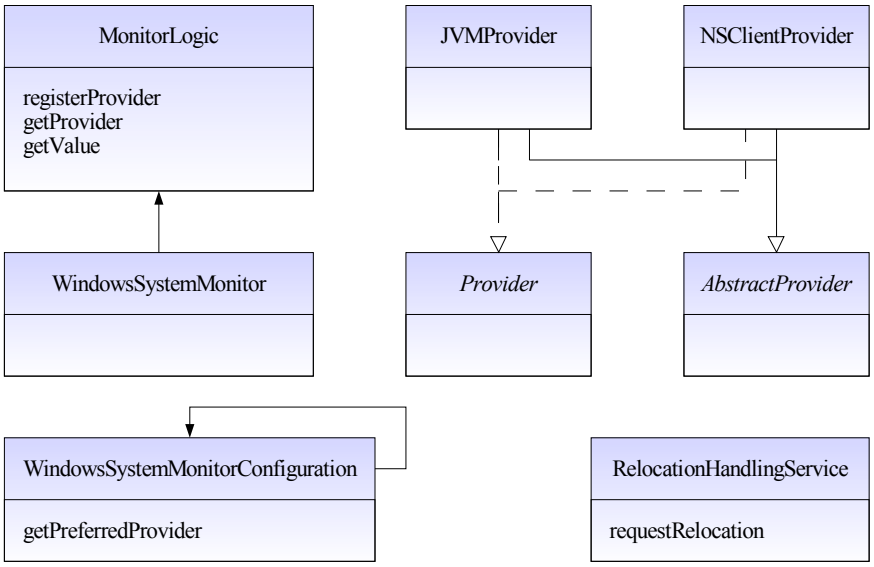

Abbildung 5.22: Klassendiagramm der Selbstoptimierungskomponenten

Für die Entscheidung, ob ein Dienst verlegt werden soll, werden Informationen über den aktuellen Systemzustand benötigt. Der `WindowsSystemMonitor` implementiert dazu das Interface `SystemMonitor`. Mit der Klasse `MonitorLogic` werden verschiedene systemabhängige Parameter zyklisch ermittelt.

Wie die Systemparameter ermittelt werden können, hängt von der zugrunde liegenden Plattform ab. Um eine Entkopplung von der realen Implementierung zu gewährleisten, wird das Interface `Provider` eingeführt, das von den systemabhängigen Klassen `JVMProvider` und `NSClientProvider` implementiert wird.

Die Klasse `JVMProvider` benutzt die Java Management Extension [39], um auf Systemen ohne nativen Zugriff auf Systemressourcen, verschiedene Werte des Systems durch die Java Virtual Machine zu ermitteln. Die Güte der ermittelten Werte schwanken teilweise stark zwischen verschiedenen JVM Implementierungen.

Um auf Windows basierten Systemen die gewünschten Systemparameter über eine native Schnittstelle ermitteln zu können, existiert die Klasse `NSClientProvider`. Der Zugriff auf die Werte verschiedener Systemressourcen erfolgt dabei über einen nativen Teil namens NSClient [42]. Darauf aufbauend existiert eine Klassenbibliothek für Java nsclient4j [28, 88], in der die nativen Methoden für den Zugriff auf die Werte der Systemressourcen in entsprechenden Java-Methoden gekapselt werden.

Die Klasse `WindowsSystemMonitorConfiguration` ermittelt anhand des Betriebssystems und den verfügbaren Providern, welcher Provider für die Abfrage der Systemparameter am besten geeignet ist. Die Methode

`getPreferredProviderName` gibt den Klassennamen des besten Providers zurück. Anhand des Klassennamens kann dann eine Instanz des Providers erstellt werden.

5.6.2 Komponenten für die Evaluierung

Für die Evaluierung der Selbstoptimierung wurden weitere Klassen zur Steuerung implementiert. In Abbildung 5.23 ist ein Teil der Klassen dargestellt, mit denen die Selbstoptimierung in der Middleware evaluiert wurde.

Der `EvaluationManagementConsoleService` ist für die Initialisierung und den Start der Selbstoptimierung zuständig. Dazu werden zunächst alle Knoten im Netz aufgefordert sich zu melden, damit die notwendige Anzahl an Diensten zum Erzeugen der Lasten berechnet werden kann.

Für die Evaluierung wird die tatsächliche Prozessorlast eines Dienstes auf dem Knoten als Optimierungskriterium der Selbstoptimierung herangezogen. Die Einstellungen bezüglich der zu generierenden Lasten erfolgt mittels der `EvaluationManagementConsoleGui`. Die Lastwerte werden an alle Knoten ver-

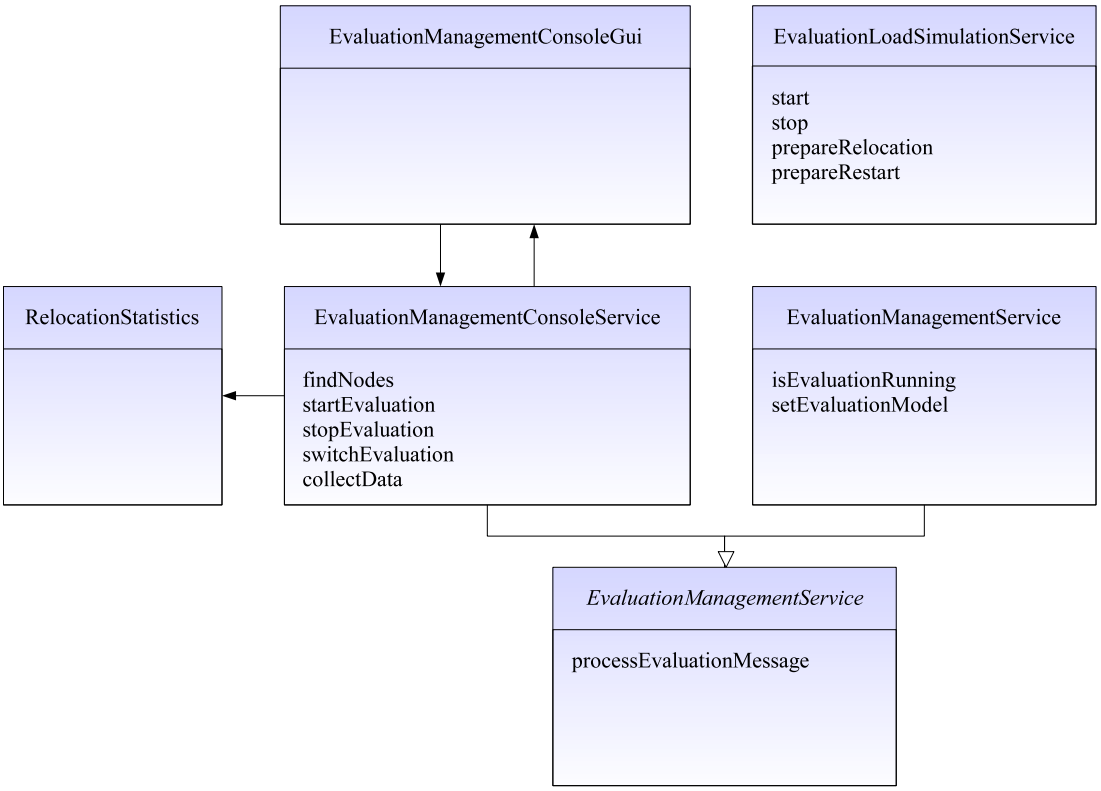

Abbildung 5.23: Klassendiagramm der Komponenten zur Steuerung der Selbstoptimierung

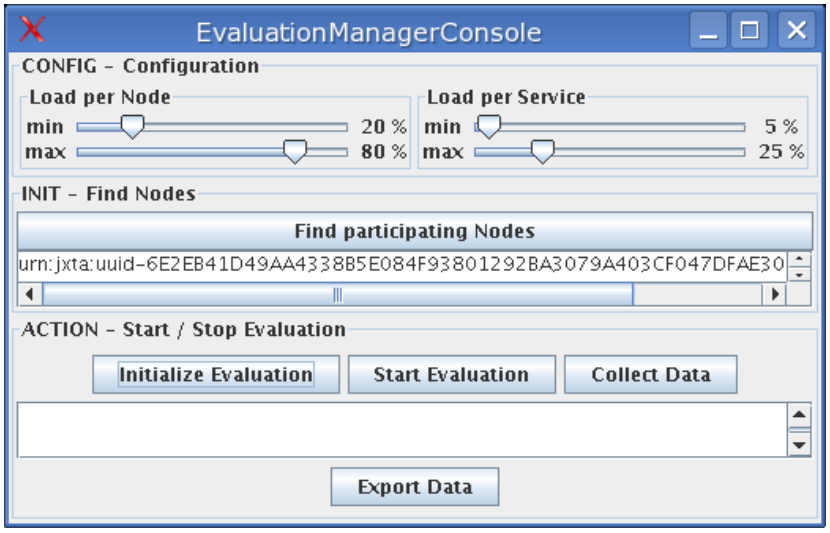

Abbildung 5.24: Grafische Oberfläche zur Steuerung der Selbstoptimierung

schickt. Der `EvaluationManagementService` erzeugt mit den gegebenen Werten eine entsprechende Anzahl von `EvaluationLoadSimulationService`-Objekten, die zwischen einem aktiven und einem passiven Zustand wechseln und zyklisch Last generieren. Die erzeugte Last wird vom `WindowsSystemMonitor` pro Dienst auf dem System beobachtet und für die Berechnung einer Dienstverlegung herangezogen.

Am Ende der Selbstoptimierung werden die Ergebnisse durch den `EvaluationManagementConsoleService` von allen beteiligten Knoten im `RelocationStatistics`-Objekt eingesammelt und in eine Excel-Datei gespeichert.

Die grafische Oberfläche der `EvaluationManagerConsoleGui` ist in Abbildung 5.24 dargestellt. Mit dieser Klasse ist es möglich, die Evaluationsparameter einzustellen. Der Parameter Load per Node bestimmt die minimale und maximale Last, die auf den Knoten generiert werden soll. Aus diesen beiden Werten wird für jeden Knoten ein zufälliger Wert für die Last eines Knotens berechnet. Der Wert Load per Service gibt die minimale und maximale Last eines Dienstes an. Für jeden Knoten werden anhand dieser Parameter solange `EvaluationLoadSimulationService`-Objekte mit einer zufälligen Last zwischen dem gegebenen Minimum und Maximum erzeugt, bis der zufällig errechnete Wert für Load per Node erreicht wurde. Mit dieser Vorgehensweise wird, ähnlich dem Simulator, für die Selbstoptimierung eine zufällige Anfangslast auf den Knoten erzeugt.

Abbildung 5.25 zeigt die grafische Oberfläche mit der die aktuelle Last auf einem Knoten visualisiert wird. Der Wert Local Load zeigt die aktuelle Gesamtlast des Knotens beziehungsweise den Verlauf der Gesamtlast im zugehörigen Diagramm

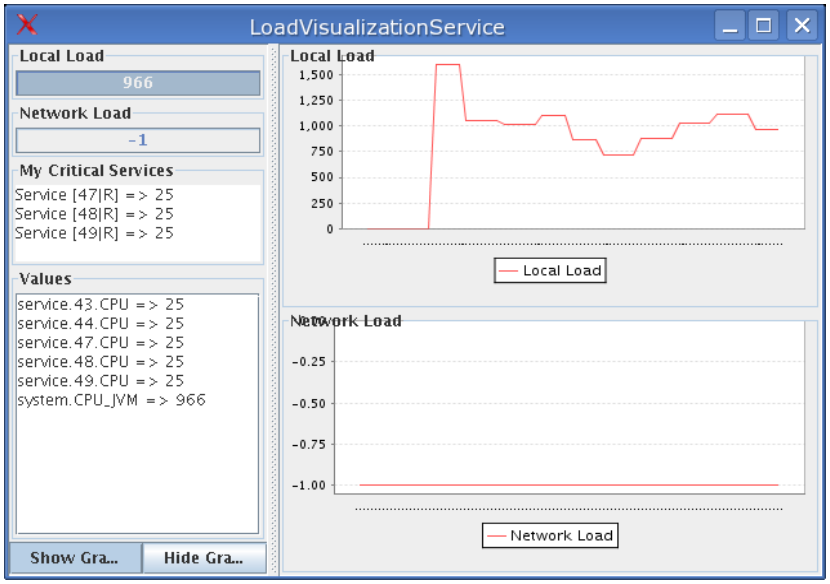

Abbildung 5.25: Grafische Oberfläche zur Anzeige der Lasten einzelner Dienste

an. Der Wert Network Load zeigt die empfangenen Lastwerte anderer Knoten an. Dieser Wert wird von der Übergabestrategie mit Hybridverfahren benötigt. Zusätzlich werden die Lasten der einzelnen Dienste in einer Liste angezeigt. Die drei Dienste, die den Knoten am stärksten belasten werden in der Liste Critical Services aufgelistet. Dies sind jedoch nicht zwangsläufig auch die Kandidaten für eine Dienstverlegung, da der zu verlegende Dienst immer so ausgewählt wird, dass er die Last zwischen den Knoten möglichst gut ausgleicht.

5.6.3 Evaluation

Die Evaluation in der Middleware wurde aufgrund der vorhandenen Computer mit zehn Knoten durchgeführt. Ziel der Evaluationen ist es zu zeigen, dass die guten Ergebnisse der Simulationen in einem realen Szenario reproduziert werden können.

Als problematisch hat sich die Messung der Lastparameter herausgestellt. Da die Middleware in Java implementiert ist, stehen nur rudimentäre Systeminformationen zur Verfügung. Aus diesem Grund wurde lediglich die CPU-Belastung herangezogen, da über den Speicherverbrauch und die Kommunikationsbandbreite keine sinnvollen Werte abgeleitet werden konnten.

Selbst die CPU-Belastung kann nur mit einer gewissen Schwankung gemessen werden, da die Dienste, die die Lastem auf den Knoten erzeugen, immer zwischen einem aktiven und einem passiven Zustand wechseln. Abhängig vom Zeitpunkt der Messung ergeben sich dadurch unterschiedliche Werte für die CPU-

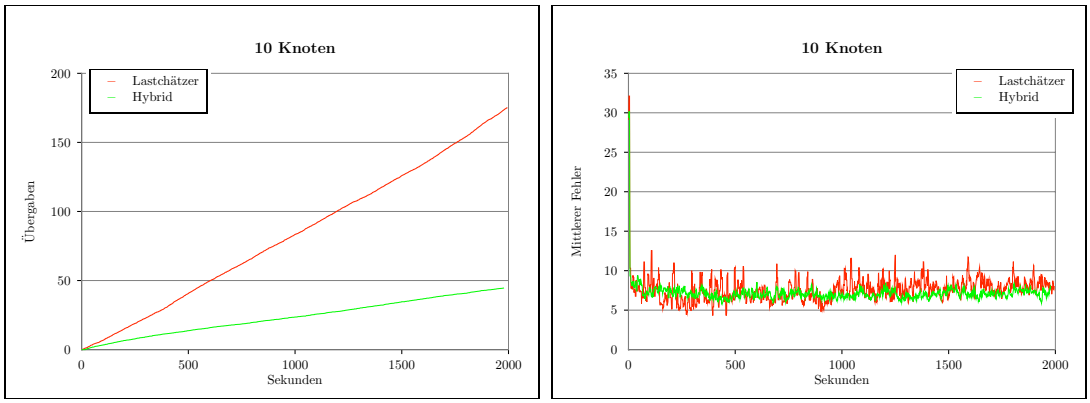

Abbildung 5.26: Anzahl der Übergaben und der mittlere Fehler für die Übergabestrategien mit Lastschätzer und Hybridverfahren bei 10 Knoten.

Belastung. Um die Schwankungen auszugleichen, wird die CPU-Belastung über einen Zeitraum von fünf Sekunden gemittelt. Die Schwankungen können jedoch nicht vollständig vermieden werden, was andererseits dem realen Verhalten dynamischer Prozesse entgegenkommt.

Evaluiert wurden die Übergabestrategie mit Lastschätzer und die Übergabestrategie mit Hybridverfahren. Beide Strategien wurden mit zehn Knoten 100-mal evaluiert. Die Parameter wurden so gewählt, dass jeder Knoten eine Last zwischen 30 und 90 Prozent erfährt. Die Lasten der Dienste wurden zwischen 10 und 25 Prozent zufällig gewählt.

Die minimale Last der Dienste wurde bewusst so groß gewählt, damit die Schwankungen bei der Messung der CPU-Belastung nicht dazu führen, dass manche Dienst keine Last mehr generieren. Durch die relativ hohe Last ist der mittlere Fehler selbstverständlich auch deutlich größer.

Jede Evaluation dauerte 2000 Sekunden wobei pro Sekunde auf jedem Knoten eine Messwertreihe aufgezeichnet wurde. Abbildung 5.26 zeigt die Diagramme mit der Anzahl der Übergaben und dem mittleren Fehler.

Bei der Anzahl der Übergaben ist für beide Übergabestrategien das gleiche Verhalten wie bei den Simulationen festzustellen. Die Übergabestrategie mit Lastschätzer erzeugt bei gleichen Einstellungen deutlich mehr Dienstverlegungen als die Übergabestrategie mit Hybridverfahren. Da die gemessenen Werte Schwankungen unterworfen sind, entspricht das Verhalten der beiden Übergabestrategien dem der Simulationen dynamischer Prozesse.

Der Lastschätzer optimiert immer, unabhängig davon, wie groß der Gewinn für die Gesamtoptimierung ist. Außerdem erkennt er nicht, dass sich die Prozesse dynamisch verhalten und nutzt somit jede Möglichkeit einen Dienst zu verlegen, was sich in einem kontinuierlichen Anstieg der Übergaben niederschlägt.

Bei der Übergabestrategie mit Hybridverfahren tritt nach etwa 1800 Sekunden ein ähnlicher Effekt wie bei den Simulationen ein. Die adaptive Schranke hat den mittleren Gewinn im System gelernt und verlegt Dienste nur noch dann, wenn der dadurch erzielte Gewinn über dem Wert der Schranke liegt. Dies drückt sich in einer leichten Abflachung der Kurve aus.

Der mittlere Fehler zeigt ein sehr gutes Verhalten für beide Strategien. Die Übergabestrategie mit Hybridverfahren ist nach 1500 Sekunden, im Vergleich zu den Simulationen, sogar etwas besser als die Übergabestrategie mit Lastschätzer. Die großen Schwankungen der Übergabestrategie mit Lastschätzer ergeben sich aus der hohen Anzahl an Dienstverlegungen und den damit verbundenen Schwankungen der Knotenbelastungen. Wird beispielsweise ein Dienst mit einer hohen CPU-Belastung verlegt, kann es dazu führen, dass durch die Verlegung beim Empfängerknoten eine Last über dem mittleren Maß entsteht und dadurch der mittlere Fehler sogar vorübergehend vergrößert wird. Außerdem spielt die Belastung durch die Verlegung der Dienste eine zusätzliche Rolle, die zu einer weiteren kurzfristigen Erhöhung der CPU-Belastung führen kann.

Bei den gewählten Grenzen für die Lasten der Dienste liegt die theoretische Belastung des mittleren Dienstes bei 17,5 Prozent. Der rechnerische mittlere Fehler würde somit bei der Hälfte des Wertes liegen. Aus dem Diagramm ist zu erkennen, dass der mittlere Fehler zwischen 8 und 9 liegt, der theoretische Wert somit sehr gut erreicht wird.

5.6.4 Betrachtung des Zusatzaufwands

Die Übergabestrategien der Selbstoptimierung nutzen unterschiedliche Informationen zur Berechnung der Lasten, um lokal entscheiden zu können, ob ein Dienst verlegt werden soll oder nicht. Die einzige Informationen, die zwischen den Knoten ausgetauscht wird ist die Lastinformation, die auf jedem Knoten lokal mit mehreren Lastparametern berechnet und auf ausgehende Nachrichten aufgeprägt wird.

Die Lastwerte sind auf den Bereich von 0 bis 100 normiert und können mit sieben Bit repräsentiert werden. Es genügt also, ein Byte pro Nachricht zusätzlich zu versenden, um die Selbstoptimierung mit den notwendigen Informationen zu versorgen. Der zusätzliche Kommunikationsaufwand ist somit minimal.

Für die Entscheidung, ob ein Dienst verlegt werden soll oder nicht, werden bei der Übergabestrategie mit Hybridverfahren die meisten Informationen benötigt. Es gibt drei Listen, in denen die Werte für die Berechnung gespeichert werden. In einer Liste werden die letzten Lastwerte anderer Knoten gespeichert, in der zweiten Liste die letzten Gewinne und in der dritten Liste die letzten berechneten Optimierungspotentiale. In den Listen werden ebenfalls ganzzahlige Werte von 0 bis 100 gespeichert. Die Listen sind auf wenige Daten begrenzt und benötigen

bei den gewählten Einstellungen der Evaluation in der Middleware lediglich 20 Byte. Zehn Byte für die Historie der Lasten und jeweils fünf Byte für die Liste der Gewinne und der Optimierungspotentiale.

Die Algorithmen sind sehr einfach und nutzen fast ausschliesslich Summation und Division zu Berechnung der arithmetischen Mittel. Lediglich zur Berechnung der adaptiven Schranke wird die Logarithmusfunktion benötigt.

Insgesamt wird durch die Selbstoptimierung ein sehr geringer Zusatzaufwand erzeugt. Es werden nur wenige Informationen über das Netzwerk verschickt und der Speicherbedarf sowie die notwendige Rechenleistung sind ebenfalls sehr gering.

5.7 Verwandte Forschungsarbeiten

Die Optimierung technischer Systeme ist Gegenstand der Forschung, seit es diese Systeme gibt. Mit dem Versuch, ein System weiter zu optimieren, können in vielen Anwendungsbereichen Ressourcen, Zeit und häufig auch Geld gespart werden, bei gleichzeitiger Steigerung der Güte des Systems.

Bereits einfache Regelkreise, wie sie aus der Regelungstechnik bekannt sind, können durch Rückkopplungen im Regelkreis, unter Verwendung von PID-Reglern ein adaptives Verhalten zeigen. Hierbei handelt es sich meist um zentral gesteuerte Systeme, deren Ziel es ist, eine Regelgröße so zu verändern, dass der Ist-Wert (tatsächliche Ausgabe) den Soll-Wert (erwünschte Ausgabe) möglichst gut erreicht.

Ein weiterer Bereich, bei dem es um die Optimierung von Systemen und im Speziellen um die Verteilung von Lasten geht, ist das so genannte Load-Balancing [59]. Bei diesen meist verteilten Systemen sollen Lasten so auf die Komponenten des Systems verteilt werden, dass die Aufgaben (Tasks), durch welche die Lasten entstehen, möglichst schnell beziehungsweise effizient verarbeitet werden. Diese Vorstellung von Last beruht auf der Annahme, dass an bestimmten Stellen in einem verteilten System Aufgaben zur Erfüllung in einer Warteschlange anstehen. Die Verteilung der Aufgaben auf die Knoten des Systems wird dann als Load-Balancing bezeichnet.

Besonders in diesem Bereich gibt es unzählige Forschungsarbeiten, die sich mit verschiedenen Strategien für die Verteilung der Aufgaben befassen. Darunter befinden sich auch Strategien, die durch eine gewisse Zufälligkeit bei der Verteilung der Aufgaben hervorragende Ergebnisse erzielen.

Der vorliegende Ansatz zur Selbstoptimierung unterscheidet sich von diesen Verfahren in zwei grundlegenden Punkten. Zum einen wird bei der Selbstoptimierung keine aktive Benachrichtigung zwischen den Knoten benutzt und zum anderen wird die Information der Lasten immer ausschließlich lokal gemessen und nur

durch Aufprägen auf die Nachrichten im System verteilt. Aus diesem Grund soll die Selbstoptimierung mit Verfahren verglichen werden, die auf ähnlichen Annahmen beruhen.

5.7.1 Reaktions-Diffusions-Modell

Alan Turing legte 1952 mit seiner Veröffentlichung „The Chemical Basis of Morphogenesis“ [77] die Grundlagen für die so genannten Reaktions-Diffusions-Modelle. Inspiriert durch ein wirbelloses Süßwassertier namens Hydra, untersuchte er die mathematischen Grundlagen zur Beschreibung der chemischen Prozesse der Morphogenese[1].

Wird eine Hydra in zwei Teile zerschnitten, ist der basale Teil mittels Morphogenese in der Lage neue Tentakeln und einen Mund zu bilden, während der Kopf einen neuen Körper ausbildet, ohne dabei neues Gewebe durch Zellteilung zu bilden. Die Erzeugung der neuen Teile geschieht ausschließlich durch Verschiebung, Umordnung und Ausdifferenzierung der vorhandenen Zellen.

Die Steuerung des Prozesses bei der Morphogenese der Hydra hat Turing mit Differenzialgleichungen beschrieben, die er aus den chemischen Prozessen abgeleitet hat, soweit sie zu diesem Zeitpunkt bekannt waren. Treibende Kraft für diesen Prozess sind dabei die Konzentrationsunterschiede chemischer Stoffe, die durch Diffusion in benachbarte Bereiche eine Reaktion in den Zellen hervorrufen.

Turing hat an mehreren Beispielen gezeigt, dass die mathematischen Grundlagen seines Reaktions-Diffusions-Modells auch auf andere natürliche Phänomene, wie beispielsweise der Blattanordnung der Waldmeisterpflanze, übertragbar ist.

Auf der Arbeit von Turing basiert sämtliche Forschung im Bereich der Reaktions-Diffusions-Modelle, wie sie beispielsweise auch bei der künstlichen Nachbildung natürlicher Muster [89] Anwendung findet. Tatsächlich arbeitet das menschliche Hormonsystem auf ähnliche Weise, jedoch wird die Verbreitung der Hormone zusätzlich durch den Blutkreislauf auf den gesamten Körper erweitert und ist somit nicht ausschließlich auf lokale Diffusionsvorgänge beschränkt.

Eine Anwendung, die der vorgestellten Selbstoptimierung ähnelt, ist das Digital Hormone Model (DHM) [58]. Das DHM beruht auf dem Reaktions-Diffusions-Modell zur Steuerung eines Roboterschwarms. Durch Veränderung der Hormondiffusion ist der Roboterschwarm in der Lage verschiedene Aufgaben zu erfüllen. Beispiele für Aufgaben sind das Ausmessen von Objekten, das Überwachen einer Region oder das Umfahren von Hindernissen.

Der Nachteil des Reaktions-Diffusions-Modells liegt in seiner Komplexität. Die notwendigen Differenzialgleichungen zur Beschreibung der Vorgänge sind sehr

[1] Morphogenese ist griechisch und bedeutet: *Die Entstehung der Form.*

komplex und rechenintensiv. Der Ansatz der Selbstoptimierung ist es, durch einfache Mechanismen zu guten Ergebnissen zu kommen. Die Differenzialgleichungen beschreiben die tatsächlichen Vorgänge möglicherweise exakter, als es mit den einfachen Berechnungen der Selbstoptimierung der Fall ist, jedoch zeigen die Evaluationsergebnisse, dass das erzielte Resultat das nahezu das Optimum erreicht und somit ein deutlich höherer Aufwand für eine minimale Verbesserung des Ergebnisses nicht gerechtfertigt ist. Darüber hinaus ist die Anpassung der Selbstorganisation auf unterschiedliche Szenarien sehr viel einfacher möglich, als der Entwurf passender Differenzialgleichungen.

5.7.2 Biologisch inspirierte Middleware

Aus dem Bereich der biologisch inspirierten Ansätze gibt es Forschungen, die sich unter anderem mit der Selbstorganisation eines Peer-to-Peer-Sensornetzwerks [29] befassen. Es wird gezeigt, dass das Netzwerk in der Lage ist, Lasten im System zu balancieren, indem ausschließlich lokale Information zur Bewertung und Verteilung der Lasten herangezogen wird. In verschiedenen Simulationen konnte damit beinahe das Optimum für die Verteilung der Last erzielt werden.

Der Unterschied zur vorgestellten Selbstoptimierung liegt darin, dass die Knoten des Sensornetzwerks eine aktive und gerichtete Kommunikation voraussetzen. Die Informationen bei der Selbstoptimierung werden auf Nachrichten aufgeprägt, bei denen der Empfänger der Nachricht nicht bekannt ist. Durch die zusätzliche Betrachtung der Kommunikationswege könnte die Anzahl der Nachrichten für die Selbstoptimierung vermutlich noch weiter gesenkt werden. Es sollte aber gerade ein Ansatz gewählt werden, der von möglichst wenig Annahmen bezüglich der vorhandene Kommunikationsinfrastruktur ausgeht.

Ein weiteres Beispiel für die Nachbildung eines natürlichen Phänomens zur Steuerung eines künstlichen Systems wird in [14] vorgestellt. Eine Rückkopplungsschleife, wie sie im menschlichen Körper zur Messung des Blutdrucks zum Einsatz kommt wird verwendet, um zu entscheiden, ob in einem Sensornetzwerk eine Aufgabe fertig gestellt ist. Ziel der Arbeit ist die Verteilung von Aufgaben in einem Sensornetzwerk, um die gestellten Aufgaben möglichst energieeffizient zu lösen.

Auch bei diesem Ansatz wird von einer aktiven Kommunikation zur Verteilung der Informationen ausgegangen. Es wird zwar der Ansatz der gerichteten Diffusion [27] benutzt, aber trotzdem sind die Empfänger einer Information a priori bekannt. Im Fall des menschlichen Hormonsystems sind die Empfänger der ausgeschütteten Hormone nicht bekannt, wie es auch für die Umsetzung der Selbstoptimierung angenommen wird.

5.7.3 Agententechnologie

Es gibt eine Reihe von Forschungsarbeiten, bei denen versucht wird, die Lastverteilung in einem System mit Hilfe von Agenten zu lösen. In [7] wird ein Ansatz zur Lastbalancierung vorgestellt, bei der eine Hierarchie von Agenten zum Einsatz kommt. Es wird zwischen lokalen Ressourcen, die bereits aus mehreren Knoten bestehen können, und globalen Ressourcen unterschieden. Jeder Agent repräsentiert eine der Ressourcen und ist für die Zuweisung der Aufgaben (Tasks) verantwortlich.

Ein weiterer Ansatz [45] versucht durch Nutzung des Reinforcement Learning die Größe der Datenpakete zu lernen, die für die Ausführung eines Dienstes benötigt werden. Die Last wird in dem vorgestellten Beispiel durch parallele Algorithmen erzeugt, die ständig mit Daten versorgt werden müssen, um arbeiten zu können. Ziel ist es, alle Knoten in einem heterogenen Netzwerk gleichmäßig auszulasten.

Das Problem der Ansätze mit Agententechnologien liegt darin, dass bei heterogenen Umgebungen für das Agentensystem selbst auch alle vorhandnen Kommunikationsmechanismen für die Kommunikation der Agenten untereinander unterstützt werden müssen, da das Agentensystem oft nicht elementarer Bestandteil der Middleware ist. Darüber hinaus besteht das Problem, dass aufgrund der aufwändigen Berechnungen für die Zuordnung der Tasks die Systeme nicht gut skalierbar sind.

Ein weiteres Problem entsteht, wenn Tasks oder Knoten hinzugefügt, beziehungsweise entfernt werden. Um ein globales Optimierungsverhalten zu erzielen, werden beispielsweise Informationen anderer Knoten gespeichert. Wird einer der Knoten entfernt, sind die resultierenden Ergebnisse falsch und es muss ein zusätzlicher Kommunikationsaufwand in Kauf genommen werden, um den aktuellen Systemzustand zu ermitteln.

Die meisten Agentensysteme arbeiten mit einer aktiven Benachrichtigung und erzeugen damit einen deutlichen Mehraufwand in Bezug auf die Kommunikation. Die Selbstoptimierung prägt den Nachrichten lediglich wenige Byte an Information auf und reduziert damit die Belastung des Systems auf ein Minimum.

Die genannten Nachteile der Ansätze mit Agententechnologie werden durch die vorgestellte Selbstoptimierung vermieden. Die Selbstoptimierung arbeitet ausschließlich lokal und das Entfernen und Hinzufügen von Knoten erfordert keinen zusätzlichen Aufwand. Selbst im Fall der adaptiven Schranke und des Hybridverfahrens, bei denen auch Informationen von anderen Knoten gespeichert werden, ist die Herkunft der Informationen irrelevant.

5.8 Fazit

Die Selbstoptimierung, mit dem menschlichen Hormonsystem als Vorbild, stellt eine sehr gute Lösung für die Optimierung eines verteilten Systems ohne zentrale Kontrolle dar. Durch das Aufprägen von Lastinformationen auf den Nachrichtenfluss der Middleware werden mit einem sehr einfachen Mechanismus nahezu optimale Ergebnisse erzielt. Darüber hinaus ist der Algorithmus beliebig skalierbar und zeigt selbst in sehr großen Netzen ein hervorragendes Optimierungsverhalten ohne zusätzlichen Anpassungsaufwand an die Netzgröße.

Die Simulationen haben das Verhalten der unterschiedlichen Übergabestrategien bei verschiedenen Netzgrößen und unterschiedlichen Parametereinstellungen gezeigt. Bereits aus den Simulationen ist zu erkennen, dass die Vorgehensweise, Informationen zufällig zu verteilen, optimal funktioniert, sofern genügend Nachrichten zwischen den Knoten ausgetauscht werden.

Die Evaluationen in der Middleware bestätigen die Ergebnisse aus den Simulationen, trotz der schwierigen Messung des Lastparameters. Obwohl die Evaluation nur in einem kleinen Netz erfolgen konnte, sind die gemessenen Resultate hervorragend und bestätigen das erkannte Verhalten aus den Simulationen. Da alle Evaluationen 100-mal durchgeführt wurden, kann ausgeschlossen werden, dass es sich bei den Messwerten zufälligerweise um sehr gute Ergebnisse handelt, die sich nicht reproduzieren lassen. Die große Bandbreite bei den Lastparametern der Knoten und der Dienste stützen diese Annahme zusätzlich.

6 Zusammenfassung und Ausblick

Die vorgestellte Architektur der Organic Ubiquitous Middleware sowie die Umsetzung der Selbstkonfiguration und Selbstoptimierung werden in diesem Kapitel zusammengefasst. Anschließend wird ein Ausblick auf mögliche Erweiterungen und Verbesserungen gegeben.

6.1 Zusammenfassung

In dieser Arbeit wurde eine Architektur für eine Middleware vorgestellt und implementiert, die für den Einsatz in ubiquitären Umgebungen konzipiert ist und die Eigenschaften zur Selbstkonfiguration und Selbstoptimierung besitzt.

6.1.1 Organic Ubiquitous Middleware

Die Architektur der Organic Ubiquitous Middleware besitzt vom Grundaufbau, mit der Unterscheidung in eine Transportschicht, eine Vermittlungsschicht und eine Diensteschicht, viel Ähnlichkeit mit herkömmlichen Middlewaresystemen, unterscheidet sich aber in wesentlichen Punkten, wie beispielsweise dem typisierten Nachrichtenaustausch, dem Monitoring auf mehreren Ebenen und dem Aufbau als Observer-Controller-Architektur. Durch diese Maßnahmen wird die Integration und Umsetzung der Selbst-X-Eigenschaften ermöglicht.

Die Anforderungen der Middleware an die zugrunde liegende Kommunikationsinfrastruktur sind minimal und durch nahezu jeden Kommunikationsmechanismus zu gewährleisten. Dies ermöglicht die Integration fast aller Geräte und damit den Einsatz in ubiquitären Umgebungen. Der Nachrichtenaustausch basiert auf einer asynchronen Kommunikation typisierter Nachrichten.

Die Transportschicht abstrahiert vollständig von der Kommunikationsinfrastruktur und kann durch einen Plug-In-Mechanismus beliebig erweitert werden. Als Beispiel wurde der JXTATransportConnector zur Anbindung des JXTA Peer-to-Peer-Netzwerks implementiert.

Die Typisierung der Nachrichten bietet bei der Zustellung der Nachrichten mehr Freiheiten als konventionelle Middlewaresysteme, die auf dem Stub-/Skeleton-Prinzip beruhen. Ändert sich bei diesen Systemen die Schnittstelle eines Dienstes ist eine Übersetzung aller abhängiger Komponenten erforderlich, die diese Funktionalität nutzen. Bei der Organic Ubiquitous Middleware kann ein Dienst jederzeit um zusätzliche Parameter erweitert werden, ohne dass andere Dienste davon betroffen sind. Abhängig von den Parametern, die in einer Nachricht übermittelt werden, kann entweder die bisherige oder die erweiterte Funktionalität des Dienstes genutzt werden.

Durch ein intensives Monitoring, sowohl auf der Transportschicht, als auch auf der Nachrichtenvermittlungsschicht, ist die Middleware in der Lage Informationen über ihren Zustand zu ermitteln. Zusätzliche Systemmonitore ermöglichen die Überwachung der Abläufe des zugrunde liegenden Systems. Mit den gewonnenen Informationen werden über einen Event-Mechanismus die Daten verarbeitet und für die Selbst-X-Eigenschaften nutzbar gemacht. Die entstandene Observer-Controller-Architektur bildet die Grundlage zur Umsetzung der Selbst-X-Eigenschaften.

Die Entwicklung von Applikationen für die Organic Ubiquitous Middleware erfordert ein Umdenken beim Design der Anwendungen. Die Aufteilung einer Applikation in Dienste wird zwar bereits von den WebServices gefordert, die freie Verteilung der Dienste auf die Knoten des Netzes ist jedoch eine zusätzliche Eigenschaft der Middleware, die zur Umsetzung der Selbst-X-Eigenschaften benötigt wird.

Der asynchrone Klassenlader, der im Rahmen dieser Arbeit entwickelt wurde, stellt eine Neuerung dar, da der Klassenlademechanismus von Java synchron abläuft. Mit dem asynchronen Klassenlader ist es möglich von einem Knoten, von dem ein Dienst verlegt wurde die Klassendateien und Ressourcen zu laden. Damit muss eine neue Applikation nur an einem beliebigen Knoten eingespielt werden und die Middleware sorgt selbstständig für die Verteilung der Klassen. Allerdings nur dann, wenn die Klassen auch benötigt werden und nicht wie bei anderen Systemen, die sämtliche Klassen auf alle Knoten kopieren, unabhängig davon ob sie benötigt werden oder nicht.

Eine weitere Besonderheit der Middleware ist die Integration der Selbstkonfiguration und Selbstoptimierung, die nachfolgend zusammengefasst werden.

6.1.2 Selbstkonfiguration

Die Selbstkonfiguration beruht auf dem kooperativen sozialen Verhalten einer Gruppe, die ein gemeinsames Ziel verfolgt. Die Dienste einer Applikation, die in einer Konfigurationsspezifikation hinterlegt sind, sollen so auf die Knoten des

Netzes verteilt werden, dass durch die Belegung eine möglichst gleichmäßige Ressourcenauslastung der Knoten entsteht.

Die Konfigurationsspezifikation, die in Form einer XML-Datei vorliegt, wird zunächst von einem Knoten an alle anderen Knoten verteilt. Anschließend werden die Bedingungen (Constraints) geprüft und jeder Knoten belegt einen Dienst, sofern die geforderten Ressourcen verfügbar sind. Die Bedingungen in der Konfigurationsspezifikation entsprechen dem mathematischen Allquantor und vereinfachen die Zuordnung von Diensten oder Monitoren zu vorhandenen Ressourcen. Beispielsweise kann damit jeder Knoten, der ein Smart-Doorplate besitzt, eine Instanz des SmartDoorplateService starten.

Anhand einer Metrik, mit der die Dienstgüte in Abhängigkeit von den vorhandenen Ressourcen und dem Ressourcenverbrauch des Dienstes berechnet wird, findet eine Zuordnung der Dienste zu den Knoten statt. Dabei wird ein vollständig dezentraler Ansatz verfolgt, bei dem jeder Knoten die Belegung eines Dienstes den anderen Knoten im Netz mitteilt. Kann ein anderer Knoten den Dienst mit einer besseren Dienstgüte erbringen, kann er die Zuordnung überschreiben.

Eine abschließende Verifikationsphase stellt die Korrektheit der gefundenen Konfiguration sicher, indem ein Knoten seine vollständige Dienstzuordnung an die anderen Knoten sendet. Alle Knoten sind damit in der Lage, ihre eigene Konfiguration zu vervollständigen, beziehungsweise Fehler in der Konfiguration zu berichtigen. Besitzt ein Knoten eine bessere Konfiguration, teilt er seine Konfiguration den anderen Knoten mit. Wird keine weitere Verbesserung verschickt, wurde eine konsistente Zuordnung der Dienste zu den Knoten gefunden.

Simulationen haben gezeigt, dass die Selbstkonfiguration in allen Fällen zu einem korrekten Ergebnis führt und dass die dafür benötigte Anzahl an Nachrichten in Abhängigkeit von der Anzahl der Dienste bestimmt werden kann. Mit 1,2 bis 1,5 Nachrichten pro Job erreicht die Selbstkonfiguration in der Simulation sehr gute Werte bei homogener Hardware. Bei heterogener Hardware, bei denen die Knoten über unterschiedliche Ressourcen verfügen, schneidet die Selbstkonfiguration in der Simulation etwas schlechter ab.

Die Evaluation in der Middleware hat gezeigt, dass die Unterscheidung zwischen homogener und heterogener Hardware im realen Fall kaum eine Rolle spielt. Außerdem wurden bei den Evaluationen in der Middleware noch bessere Ergebnisse als bei den Simulationen erzielt.

Die Selbstkonfiguration ist durch den einfachen Mechanismus und den dezentralen Ansatz gut skalierbar und zeigt im Vergleich zu anderen Verfahren, wie beispielsweise Agentensystemen, ein deutlich besseres Verhalten mit steigender Knotenzahl.

6.1.3 Selbstoptimierung

Das menschliche Hormonsystem schüttet seine Informationen, die Hormone, in den Blutkreislauf aus und beeinflusst damit andere Zellen im Körper, die über entsprechende Rezeptoren verfügen. Die Selbstoptimierung arbeitet nach diesem Prinzip, indem Informationen über lokale Lastparameter an andere Knoten weitergegeben werden. Die Knoten entscheiden anhand der empfangenen Informationen, ob ein Dienst zur Optimierung der Last verlegt werden soll oder nicht. Ziel der Selbstoptimierung ist die gleichmäßige Belastung der Knoten des Netzes.

Mit einem Simulator wurden vier Übergabestrategien getestet und auf die Anzahl der notwendigen Dienstverlegungen hin untersucht. Ein zusätzliches Kriterium ist der mittlere Fehler, der angibt, wie gleichmäßig die Last zwischen den Knoten verteilt wurde.

Die Simulationen haben gezeigt, dass die Übergabestrategie mit Hybridverfahren, die sich aus einer Kombination der Übergabestrategie mit Lastschätzer und der Übergabestrategie mit adaptiver Schranke zusammengesetzt, die besten Ergebnisse liefert. Die Betrachtung dynamischer Prozesse zeigt die Stärken der adaptiven Schranke, die in der Lage ist, den mittleren Gewinn zu ermitteln und dann nur noch Dienstverlegungen zuzulassen, die über dieser Schranke liegen. Damit wird eine Reduzierung der Dienstverlegungen teilweise um mehr als 80 Prozent gegenüber der einfachen Übergabestrategie erzielt.

Die Evaluationen in der Middleware zeigen auch hier noch bessere Ergebnisse als bei den Simulationen. Ein Netzwerk mit 10 Knoten wurde in 100 Evaluationen mit der Übergabestrategie mit Lastschätzer und der Übergabestrategie mit Hybridverfahren getestet. Die Ergebnisse zeigen, dass die Übergabestrategie mit Hybridverfahren im realen Fall sogar bessere Werte erzielt als die Übergabestrategie mit Lastschätzer, die zwar mehr Dienstverlegungen erzeugt, in den Simulationen dafür aber einen kleineren mittleren Fehler erreichte. Bei den Evaluationen in der Middleware zeigten sich sowohl für die Anzahl der Übergaben, als auch beim mittleren Fehler bessere Werte für die Übergabestrategie mit Hybridverfahren.

6.2 Ausblick

Aufbauend auf dieser Arbeit und den daraus gewonnenen Erfahrungen ergeben sich eine Reihe neuer Entwicklungsmöglichkeiten, sowohl auf der Ebene der Middleware, als auch bei der Umsetzung und Integration der Selbst-X-Eigenschaften.

Middleware

Bei der Umsetzung der Referenzanwendung, dem Smart-Doorplate-Projekt, aber auch bei der Implementierung der Selbst-X-Eigenschaften hat sich gezeigt, dass

der Mechanismus der typisierten Nachrichten sehr mächtig, aber auch mit einem höheren Aufwand in der Implementierung verbunden ist. Eine zusätzliche Schicht, die einen Methodenaufruf kapselt, wäre wünschenswert.

Da alle Dienste als Threads in der Middleware gestartet werden, hat sich die Messung der Systemparameter als sehr aufwändig und unzuverlässig herausgestellt. Außerdem ist Java nicht in der Lage einen Thread durch einen anderen Thread zu beenden, was besonders bei der Umsetzung des Selbstschutzes von Bedeutung ist. Somit könnte ein bösartiger Dienst nicht beendet werden. An dieser Stelle soll die Middleware auf die Verwendung von Prozessen erweitert werden, die beispielsweise über einen gemeinsamen Speicher (Shared-Memory) kommunizieren können.

Um die Middleware für eine breite Nutzung zugänglich zu machen, müssen weitere TransportConnectoren implementiert werden, die verschiedene Kommunikationstechnologien unterstützen und somit die Integration in ein ubiquitäre Umgebung weiter vereinfachen.

Selbst-X-Eigenschaften

Die Selbstkonfiguration wird bisher nur für die initiale Konfiguration einer Applikation genutzt. Die Ausnutzung der Rekonfigurationsmöglichkeit, die hinter diesem Mechanismus steckt, kann ebenso für eine Selbstkonfiguration durch sich verändernde Umgebungen genutzt werden. Darüber hinaus könnte die Konfigurationssprache um zusätzliche Konstrukte erweitert werden, mit denen zusätzliche Abhängigkeiten zwischen Diensten oder den Knoten beschrieben werden könnten.

Die Selbstoptimierung stellt mit der Übergabestrategie mit Hybridverfahren bereits einen mächtigen Mechanismus zur Optimierung des Netzes auch bei dynamischem Prozessverhalten zur Verfügung. Um eine allgemeine Nutzbarkeit der Selbstoptimierung zu ermöglichen, muss ein Weg gefunden werden anwendungsspezifische Parameter in die Bewertung mit einfließen zu lassen und widersprüchliche Aussagen bei der Bewertung des Optimierungsziels zu vermeiden.

Weitere Möglichkeiten ergeben sich in der Erweiterung der Middleware um zusätzliche Selbst-X-Eigenschaften, wie beispielsweise der Selbstheilung oder des Selbstschutzes. In angrenzenden Forschungsvorhaben am Lehrstuhl wird im Rahmen des Schwerpunktprogramms 1183 der DFG „Organic Computing“ an den Grundlagen für die Umsetzung dieser Mechanismen geforscht.

Anwendungsfelder

Die Organic Ubiquitous Middleware wurde mit dem Fokus auf ubiquitäre Umgebungen Entworfen. In diesem Bereich sind die bevorzugten Anwendungsgebiete zu suchen. Das Smart-Doorplate-Projekt zeigt eine prototypische Anwendung, die

um zusätzliche Szenarien erweitert werden kann. Weitere Einsatzfelder könnten Smart-Environments wie das Smart-Home, Smart-House oder ein Smart-Office sein.

Ein weiterer Ansatzpunkt wäre die Übertragung der Middleware auf eingebettete Systeme, wie beispielsweise dem Automobil, um dort die Möglichkeiten der Selbst-X-Eigenschaften nutzbar zu machen.

Ein besonders interessantes Einsatzgebiet ergibt sich aus dem Bereich des Krisenmanagements. Bei Naturkatastrophen besteht die Notwendigkeit, sehr schnell Hilfe zu leisten, deren Güte meist von den verfügbaren Informationen abhängt. Mit einem System, das sich selbst konfiguriert und bezüglich der verfügbaren Ressourcen selbst optimiert und somit die Kommunikations- und Informationswege unterstützt, könnte die Hilfe vor Ort zielgerichteter und effizienter erfolgen.

Literaturverzeichnis

[1] Christian Becker, Markus Handte, Gregor Schiele, and Kurt Rothermel. PCOM - A Component System for Pervasive Computing. In *2nd IEEE International Conference on Pervasive Computing and Communication (PerCom '04)*, Orlando, USA, 2004.

[2] Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel. BASE - A Micro-Broker-Based Middleware for Pervasive Computing. In *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Roxana Belecheanu, Gawesh Jawaheer, Asher Hoskins, Julie A. McCann, and Terry Payne. Semantic web meets autonomic ubicomp. In *ISWC'04 Workshop on Semantic Web Technology for Mobile and Ubiquitous Applications*, Hiroshima, Japan, November 2004.

[4] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[5] Raouf Boutaba, Salima Omari, and Ajay Pal Singh Virk. 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.

[6] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial Continued: The Rest of the JDK*. Addison-Wesley Professional, December 1998.

[7] Junwei Cao, Daniel P. Spooner, Stephen A. Jarvis, and Graham R. Nudd. Grid load balancing using intelligent agents. *Future Generation Computer Systems*, 21(1):135–149, 2005.

[8] Renato Cerqueira, Carlos Cassino, and Roberto Ierusalimschy. Dynamic component gluing across different componentware systems. In *International Symposium on distributed Objects and Applications (DOA '99)*, pages 362 – 371, Edinburgh, 1999. IEEE Press.

[9] The OWL Services Coalition. Semantic markup for webservices (owl-s). http://www.daml.org/services/owl-s/1.1, November 2004.

[10] Peter Cramton, Yoav Shoham, and Richard Steinberg. *Combinatorial Auctions*. MIT Press, January 2006.

[11] Michael L. Dertrouzos. The future of computing. *Scientific American*, 1999.

[12] Mark d'Inverno and Michael Luck. *Understanding Agent Systems*. Springer Series on Agent Technology. Springer, 2nd rev. and extended edition, 2004.

[13] Xiangdong Dong, Salim Hariri, Lizhi Xue, Huoping Chen, Ming Zhang, Sathija Pavuluri, and Soujanya Rao. Autonomia: An autonomic computing environment. In *International Performance Computing and Communications Conference (IP-CCC)*, pages 61–68, Phoenix, AZ, USA, April 2003.

[14] Falko Dressler, Bettina Krüger, Gerhard Fuchs, and Reinhard German. Self-organization in sensor networks using bio-inspired mechanisms. In *ARCS'05: Workshop Self-Organization and Emergence*, pages 139–144, March 2005.

[15] Partha S. Dutta, Nicholas R. Jennings, and Luc Moreau. Adaptive distributed resource allocation and diagnostics using cooperative information sharing strategies. In *Proceedings of the 5th International Conference on Autonomous Agents and Multi-Agent Systems*, Hakodate, Japan, 2006.

[16] Guy Eddon and Henry Eddon. *Inside Distributed Com*. Microsoft Press, February 1998.

[17] Inc. Enterprise Management Associates. Practical autonomic computing: Roadmap to self managing technology. http://www-03.ibm.com/autonomic/pdfs/AC_Practical_Roadmap_Whitepaper.pdf, 2006.

[18] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Pearson Education, June 1999.

[19] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Towards Distraction-free Pervasive Computing. *IEEE Pervasive Computing, special issue on Integrated Pervasive Computing Evironments*, 1(2):22–31, 2002.

[20] Francis Heylighen. The science of self-organization and adaptivity. In *The Encyclopedia of Life Support Systems*. EOLSS Publishers Co. Ltd., 2003.

[21] Tony Hoare and Robin Milner. Grand Challenges in Computing Research. http://www.ukcrc.org.uk/gcresearch.pdf, March 2004.

[22] John H. Holland. *Emergence: From Chaos to Order*. Perseus Books Group, September 1999.

[23] Florian Horn, Gerd Lindenmeier, and Isabelle Moc. *Biochemie des Menschen*. Georg Thieme Verlag, Stuttgart, 2002.

[24] Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. http://www.research.ibm.com/ autonomic/, October 2001.

[25] IBM Corporation. Autonomic computing concepts. http://www.ibm.com/ autonomic/, 2001.

[26] IEEE Foundation for Intelligent Physical Agents. Standard FIPA specifications. http://www.fipa.org/repository/standardspecs.html, 2006.

[27] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67, New York, NY, USA, 2000. ACM Press.

[28] java.net. nsclient4j. https://nsclient4j.dev.java.net/, 2005.

[29] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems*, Lecture Notes in Artificial Intelligence, pages 265–282. Springer, 2004.

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

[31] Robert Klaus. Selbstkonfiguration in einem dienstbasierten Peer-to-Peer Netzwerk. In *Diplomarbeit*, 2006.

[32] Bhaskar Krishnamachari, Stephen Wicker, Ramón Béjar, and Cèsar Fernàndez. On the complexity of distributed self-configuration in wireless networks. *Telecommunication Systems, Special Issue on Wireless Networks and Mobile Computing*, pages 169–177, 2000.

[33] Thomas Ledoux. OpenCorba: A Reflective Open Broker. In *2nd International Conference on Reflection (Reflection'99)*, pages 197–214, Saint-Malo, France, 1999.

[34] George G. Lendaris. On the definition of self-organizing systems. In *Proceedings of the IEEE*, Santa Barbara, CA, March, 1964. Defense Research Laboratories General Motors Corporation.

[35] Michael Luck, Peter McBurney, Onn Shehory, and Steve Willmott. Agent Technology Roadmap. http://www.agentlink.org/roadmap/al3rm.pdf, 2006.

[36] Alan K. Mackworth. Constraint satisfaction. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 285–293. Wiley Interscience, 1992.

[37] Alan K. Mackworth and Eugene C. Freuder. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 59:57–62, 1993.

[38] Ian Marshall. Technology inspired by nature. http://www.btexact.com/ideas/features?doc=12034, December 2002.

[39] Sun Microsystems. Java management extensions (jmx). http://java.sun.com/products/JavaManagement/, 2005.

[40] Pragnesh Jay Modi, Hyuckchul Jung, Milind Tambe, Wei-Min Shen, and Shriniwas Kulkarni. Dynamic distributed resource allocation: A distributed constraint satisfaction approach. In John-Jules Meyer and Milind Tambe, editors, *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, pages 181–193, 2001.

[41] Michael C. Mozer. The Neural Network House: An Environment that Adapts to its Inhabitants. In *AAAI Spring Symposium on Intelligent Environments, Menlo Park, CA*, pages 110–114, 1998.

[42] Nagios. Nsclient: Netsaint windows client. http://nsclient.ready2run.nl/, 2005.

[43] Object Management Group (OMG). Object Constraint Language. available online: www.omg.org/docs/ptc/03-10-14.pdf, 2003.

[44] Object Management Group (OMG). The common object request broker: Architecture and specification. revision 3.0. http://www.omg.org/technology/documents/formal/corba_2.htm, 2004.

[45] Johan Parent, Katja Verbeeck, and Jan Lemaire. Adaptive load balancing of parallel applications with reinforcement learning on heterogeneous networks. In *Int. Symposium DCABES*, Wuxi, China, December 2002.

[46] Jan Petzold. Einsatz von Sprungvorhersagetechniken zur Kontextvorhersage in ubiquitären Systemen. Technical Report 2003-4, Institut für Informatik, Universität Augsburg, March 2003.

[47] Jan Petzold. Augsburg Indoor Location Tracking Benchmarks. Context Database, Institute of Pervasive Computing, University of Linz, Austria. http://www.soft.uni-linz.ac.at/Research/Context_Database/index.php, January 2005.

[48] Jan Petzold, Faruk Bagci, Wolfgang Trumler, and Theo Ungerer. Confidence Estimation of the State Predictor Method. In *2nd European Symposium on Ambient Intelligence*, pages 375–386, Eindhoven, Niederlande, November 2004.

[49] Jan Petzold, Faruk Bagci, Wolfgang Trumler, Theo Ungerer, and Lucian Vintan. Global State Context Prediction Techniques Applied to a Smart Office Building. In *The Communication Networks and Distributed Systems Modeling and Simulation Conference*, San Diego, USA, January 2004.

[50] Project JXTA. http://www.jxta.org, November 2002.

[51] Karl Rüdiger Reischuk. *Komplexitätstheorie Band 1: Grundlagen.* Teubner Verlag, 1999.

[52] Manuel Román and Roy H. Campbell. GAIA: Enabling Active Spaces. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 229–234, Kolding, Denmark, September 2000.

[53] Manuel Román, Christopher K. Hess, Anand Ranganathan, Pradeep Madhavarapu, Bhaskar Borthakur, Prashant Viswanathan, Renato Cerqueira, Roy H. Campbell, and M. Dennis Mickunas. GaiaOS: An Infrastructure for Active Spaces. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Avenue, Urbana, IL 61801-2987 USA, May 2001.

[54] Manuel Román, Fabio Kon, and Roy H. Campbell. Design and Implementation of Runtime Reflection in Communication Middleware: The DynamicTAO Case. In *Proceedings of the 1999 ICDCS Workshop on Electronic Commerce and Web-Based Applications*, pages 122–127, Los Alamitos, CA, USA, 1999.

[55] Manuel Román, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online Journal, Special Issue on Reflective Middleware*, July 2001.

[56] MIT Computer Science and Artificial Intelligence Laboratory. MIT Project OXYGEN: Pervasive Human-Centered Computing. http://www.oxygen.lcs.mit.edu/index.html, 2004.

[57] Onn Shehory and Sarit Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1–2):165–200, 1998.

[58] Wei-Min Shen, Peter Will, Aram Galstyan, and Cheng-Ming Chuong. Hormone-inspired self-organization and distributed control of robotic swarms. *Autonomous Robots*, 17:93–105, 2004.

[59] Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson. *Scheduling and Load Balancing in Parallel and Distributed Systems*. Wiley-IEEE Computer Society Press, 1995.

[60] Stefan Silbernagl and Agamemnon Despopoulos. *Taschenatlas der Physiologie*. Georg Thieme Verlag und Deutscher Taschenbuch Verlag, Stuttgart, 2001.

[61] Mark Sims, Claudia V. Goldman, and Victor Lesser. Self-organization through bottom-up coalition formation. In *AAMAS '03: Proceedings of the second International joint Conference on Autonomous Agents and Multiagent Systems*, pages 867–874, New York, NY, USA, 2003.

[62] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transaction on Computers*, 29(12):1104–1113, 1980.

[63] Sun Microsystems. Java Remote Method Invocation Specification. Revision 1.8. available online: http://java.sun.com/j2se/1.4/docs/guide/rmi/, 2002.

[64] Sun Microsystems. The java tutorial. http://java.sun.com/docs/books/tutorial/extra/regex/index.html, 2005.

[65] Sun Microsystems. Java rmi specification. http://java.sun.com/products/jdk/rmi/reference/docs/index.html, 2006.

[66] Andrew S. Tanenbaum and Maarten van Steen. *Verteilte Systeme: Grundlagen und Paradigmen*. Pearson Studium, 2002.

[67] Tobias Thiemann. Selbstorganisation der Knoten einer Peer-to-peer-Middleware mittels Botenstoffen. In *Diplomarbeit*, 2005.

[68] Martin Trepel. *Neuroanatomie Struktur und Funktion*. Urban und Fischer Verlag, 3rd edition, October 2003.

[69] Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer. Smart Doorplate. *Personal and Ubiquitous Computing*, 7(3-4):221–226, 2003.

[70] Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer. Smart Doorplate - Toward an Autonomic Computing System. In *The Fifth Annual International Workshop on Active Middleware Services (AMS2003)*, pages 42–47, Seattle, USA, June 2003.

[71] Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer. AMUN - An Autonomic Middleware for the Smart Doorplate Project. In *UbiSys '04 - System Support for Ubiquitous Computing Workshop*, Nottingham, England, September 2004.

[72] Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer. AMUN - autonomic middleware for ubiquitous environments applied to the smart doorplate. *ELSEVIER Advanced Engineering Informatics*, 19(3):243–252, 2005.

[73] Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer. AMUN - Autonomic Middleware for Ubiquitious eNvironments applied to the Smart Doorplate Project. In *International Conference on Autonomic Computing (ICAC-04)*, pages 274–275, New York, NY, May 17-18 2004.

[74] Wolfgang Trumler, Robert Klaus, and Theo Ungerer. Self-configuration vai cooperative social behavior. In *3rd IFIP International Conference on Autonomic and Trusted Computing (ATC-06)*, Wuhan, China, September 2006.

[75] Wolfgang Trumler, Tobias Thiemann, and Theo Ungerer. An artificial hormone system for self-organization of networked nodes. In *IFIP Conference on Biologically Inspired Cooperative Computing*, Santiago de Chile, August 2006.

[76] Edward. P. K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London and San Diego, 1993.

[77] Alan M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, 237(641):37–72, August 1952.

[78] Francisco J. Varela, Humberto R. Maturana, and R. Uribe. Autopoiesis: The organization of living systems, its characterization and a model. *BioSystems*, 5:187 – 196, 1974.

[79] VDE/ITG/GI. Organic Computing: Computer- und Systemarchitektur im Jahr 2010. http://www.gi-ev.de/download/VDE-ITG-GI-Positionspapier Organic Computing.pdf, October 2003.

[80] World Wide Web Consortium (W3C). Webservice Description Language. http://www.w3.org/TR/wsdl, March 2001.

[81] World Wide Web Consortium (W3C). Web service activity. http://www.w3.org/2002/ws/, January 2002.

[82] World Wide Web Consortium (W3C). Soap version 1.2 part 1: Messaging framework. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/, June 2003.

[83] World Wide Web Consortium (W3C). OWL Web Ontology Language, W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-features/, 2004.

[84] World Wide Web Consortium (W3C). Extensible markup language (xml). http://www.w3.org/XML/, February 2006.

[85] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[86] Mark Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–110, September 1991.

[87] Gerhard Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, July 2000.

[88] Nicholas Whitehead. Access windows performance monitor counters from java. http://www.javaworld.com/javaworld/jw-11-2004/jw-1108-windowspm.html, November 2004.

[89] Andrew Witkin and Michael Kass. Reaction-diffusion textures. *Computer Graphics*, 25(4):299–308, 1991.

[90] Michael Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons (Chichester, England), 2002.

[91] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998.

[92] Scott M. Zhayer and Peter Steenkist. An Architecture for the Integration of Physical and Informational Spaces. *Personal Ubiquitous Computing*, 7(2):82–90, 2003.

Abbildungsverzeichnis

Tabellenverzeichnis

A XML-Schemadefinition der Konfiguration

```
<?xml version="1.0"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

  <xs:complexType name="valueType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="name" type="xs:string" use="required" />
        <xs:attribute name="unit" type="xs:string" use="required" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:complexType name="ressource">
    <xs:sequence>
      <xs:element name="value" type="valueType" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required" />
    <xs:attribute name="uri" type="xs:string" use="required" />
  </xs:complexType>

  <xs:complexType name="base_service" abstract="true">
    <xs:attribute name="id" type="xs:positiveInteger" use="required"/>
    <xs:attribute name="amount" type="xs:positiveInteger" use="required" />
    <xs:attribute name="class" type="xs:string" use="required" />
    <xs:attribute name="name" type="xs:string" use="required" />
  </xs:complexType>
```

```
<xs:complexType name="service">
  <xs:complexContent>
    <xs:extension base="base_service">
      <xs:sequence>
        <!-- requires -->
        <xs:element name="monitor" type="monitor" minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element name="ressource" type="ressource" minOccurs="0"
                    maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="monitor">
  <xs:complexContent>
    <xs:extension base="base_service">
      <xs:sequence>
        <xs:element name="ressource" type="ressource" minOccurs="0"
                    maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="having">
  <xs:sequence>
    <xs:element name="ressource" type="ressource" minOccurs="0"
                maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="provide">
  <xs:sequence>
    <xs:element name="monitor" type="monitor" minOccurs="0"
                maxOccurs="unbounded"/>
    <xs:element name="service" type="service" minOccurs="0"
                maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

```
  <xs:complexType name="forall">
    <xs:sequence>
      <xs:element name="having" type="having" />
      <xs:element name="provide" type="provide" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="constraints">
    <xs:sequence>
      <xs:element name="forall" type="forall" minOccurs="1" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="minimumConfiguration">
    <xs:sequence>
      <xs:element name="service" type="service" minOccurs="0"
                  maxOccurs="unbounded" />
      <xs:element name="monitor" type="monitor" minOccurs="0"
                  maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="optionalConfiguration">
    <xs:sequence>
      <xs:element name="service" type="service" minOccurs="0"
                  maxOccurs="unbounded" />
      <xs:element name="monitor" type="monitor" minOccurs="0"
                  maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="configuration">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="constraints"
                    type="constraints" minOccurs="0" />
        <xs:element name="minimumConfiguration"
                    type="minimumConfiguration"  minOccurs="1" />
        <xs:element name="optionalConfiguration"
                    type="optionalConfiguration"  minOccurs="0" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

B Eigene Veröffentlichungen

2006

Wolfgang Trumler, Robert Klaus, and Theo Ungerer.
Self-configuration vai cooperative social behavior. In *3rd IFIP International Conference on Autonomic and Trusted Computing (ATC-06)*, Wuhan, China, September 2006.

Wolfgang Trumler, Tobias Thiemann, and Theo Ungerer.
An artificial hormone system for self-organization of networked nodes. In *IFIP Conference on Biologically Inspired Cooperative Computing*, Santiago de Chile, August 2006.

Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer.
AMUN – An Autonomic Middleware for the Smart Doorplate Project. *Journal of Personal and Ubiquitous Computing*, volume 10, number 7-11, 2006

2005

Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer.
AMUN – autonomic middleware for ubiquitous environments applied to the smart doorplate. *ELSEVIER Advanced Engineering Informatics*, 19(3):243 – 252, 2005.

2004

Wolfgang Trumler, Jan Petzold, Faruk Bagci, and Theo Ungerer.
Towards an Organic Middleware for the Smart Doorplate Project. *GI Workshop on Organic Computing - in connection with „34. Jahrestagung der Gesellschaft für Informatik“*, pages 626 – 630, Ulm, Germany, September 24, 2004.

Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer.
AMUN – An Autonomic Middleware for the Smart Doorplate Project. In *UbiSys 2004 - System Support for Ubiquitous Computing Workshop*, Nottingham, England, September 2004.

Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer.
AMUN – Autonomic Middleware for Ubiquitious eNvironments applied to the Smart Doorplate Project. In *International Conference on Autonomic Computing (ICAC-04)*, pages 274 – 275, New York, NY, May 2004.

2003

Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer.
Smart Doorplate – Toward an Autonomic Computing System. In *The Fifth Annual International Workshop on Active Middleware Services (AMS2003)*, pages 42 – 47, Seattle, USA, June 2003.

Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer.
Smart Doorplate. In *The First International Conference on Appliance Design (1AD)*, Bristol, UK, May 6-8, 2003, Reprinted in: *Journal of Personal and Ubiquitous Computing*, volume 7, number 3-4, pages 221 – 226, July 2003