Fachhochschule Wiesbaden
Fachbereich Design Informatik Medien
Studiengang Informatik

Master-Thesis
zur Erlangung des akademischen Grades
Master of Science – M.Sc.

Bewertung von Selbstorganisationsmechanismen für Managementkomponenten auf Basis von Simulationen und Leistungsmessungen

vorgelegt von Bernhard Jungk

am 30. September 2008

Referent: Prof. Dr. Reinhold Kröger
Korreferent: Prof. Dr. Steffen Reith

Erklärung gem. BBPO, Ziff. 6.4.2

Ich versichere, dass ich die Master-Thesis selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Wiesbaden, 30.09.2008 Bernhard Jungk

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Master-Thesis:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Bibliothek der FHW	$\surd$	
Veröffentlichung des Titels der Arbeit im Internet	$\surd$	
Veröffentlichung der Arbeit im Internet	$\surd$	

Wiesbaden, 30.09.2008 Bernhard Jungk

Inhaltsverzeichnis

1 Einführung **1**

2 Grundlagen **5**

2.1 Service Level Management 5
2.2 Selbstorganisierende Systeme 6
2.2.1 Klassifizierung 7
2.2.2 Autonomic Computing 7
2.2.3 Beispiele 8
2.3 Algorithmisches Mechanismus-Design 9
2.3.1 Spieltheorie 10
2.3.2 Einmalige und wiederholte Spiele 11
2.3.3 Agenten-Typen 12
2.3.4 Modell 14
2.3.5 Eigenschaften eines Mechanismus 15
2.3.6 Gleichgewichts-Strategien 17
2.3.7 Komplexität 18
2.3.8 Vickrey-Clarke-Groves Mechanismen 19
2.3.9 Auktionen 20
2.4 Gruppenkommunikation 24
2.4.1 Übersicht 24
2.4.2 Mitglieder-Service 25
2.4.3 Nachrichtenauslieferung 25
2.4.4 Nachrichtenreihenfolge 27
2.4.5 Lebendigkeit 27
2.4.6 Beispiel - JGroups 28

3 Analyse **37**

3.1 Problemstellung 37
3.2 Systemdefinition 38
3.3 Dienstgüte-Eigenschaften 41
3.4 Lösungsansätze 43
3.4.1 Lockerung des SLOs 44
3.4.2 Verkürzung der Antwortzeit 47
3.4.3 Kombination beider Verfahren. 49
3.5 Das SLO-Spiel 50
3.5.1 Mechanismus 50
3.5.2 Eigenschaften 58
3.5.3 Komplexität 60
3.5.4 Verallgemeinertes SLO-Spiel 62
3.5.5 Offene theoretische Fragestellungen 66
3.6 Leistungsmessung und -Bewertung 67

4 Entwurf **71**

4.1 Architektur 71
4.2 Detailentwurf 77
4.2.1 Service 77
4.2.2 Service-Manager 78
4.2.3 Strategie 80
4.2.4 Simulations-Controller 86
4.2.5 Verteiltes System 88
4.3 Informationsmodell 88
4.3.1 Konfigurations- und Steuerungsinformationen 88
4.3.2 Trace-Log 89

5 Implementierung **91**

5.1 Konfiguration 91
5.2 Auswertung 96
5.3 JMX 97
5.4 Gruppenkommunikation und JGroups 99
5.5 Grafische Benutzeroberfläche 102
5.6 Codestatistiken 104

6 Auswertung **105**
6.1 Bewertung des Einschwingvorgangs 105
6.2 Verhalten während dynamischer Änderungen 114
6.3 Verhalten bei aktiviertem Scheduling 118

7 Zusammenfassung **121**

A Literaturverzeichnis **125**

Abbildungsverzeichnis

2.1 Die Darstellung des Gefangenendilemmas in Matrix-Form. 10
2.2 Die Architektur von JGroups. 29

3.1 Ein Beispielworkflow mit mehreren Sequenzen, einer Parallelausführung und einer Schleife. 40
3.2 Der identische Workflow wie in Abbildung 3.1 in einer anschaulicheren Darstellung. 40
3.3 Die Transformation des Workflows aus Abbildung 3.2 in eine Baumstruktur. 45
3.4 Scheduling-Modell eines Services mit idealer Antwortzeit von 10 ms. . . 48
3.5 Zwei Aktivitäten die den gleichen Service nutzen. 49
3.6 Sequenz 2 teilt sich zwei Services mit jeweils einer anderen Sequenz. . . 49

4.1 Die Architektur des Systems. 72
4.2 Gruppenbildung mehrerer Strategien. 73
4.3 Die Architektur des Systems mit Simulations-Controller. 74
4.4 Die Interaktion des Simulations-Controller. 74
4.5 Die Architektur des Systems als verteiltes System. 77
4.6 Schnittstelle und Implementierung des Services. 78
4.7 Klassendiagramm des Service Managers. 78
4.8 Der grundsätzliche Ablauf einer Management-Aktion. 79
4.9 Die Struktur des Thread-Pools. 79
4.10 Das Aktivitätsdiagramm zum Thread-Pool. 80
4.11 Die übergeordneten Zustände eines Agenten. 81
4.12 Das Zustandsmodell eines Verkäufers. 82
4.13 Das Zusstandsmodell eines Käufers. 83
4.14 Der Aufbau des `ParallelAggregators`. 85
4.15 Die Erzeugung eines Services. 87

4.16 Veränderungen an der Simulation. 87

5.1 Gruppenbildung mehrerer Strategien. 96
5.2 Struktur des Multicast-RPC-Diensts mit JGroups. 101
5.3 Die Konfigurationsoberfläche des Simulations-Controllers. 103
5.4 Die Überwachungsoberfläche des Simulations-Controllers. 103

6.1 Einschwingverhalten aus Sicht der Bieter in der verteilten Messung mit 10 Agenten. 106
6.2 Einschwingverhalten aus Sicht der Bieter in der Simulation mit JGroups mit 10 Agenten. 107
6.3 Einschwingverhalten aus Sicht der Bieter in der Simulation mit Ersatz für JGroups mit 10 Agenten. 107
6.4 Der Kommunikationsaufwand des Einschwingverhaltens in der verteilten Messung mit 10 Agenten. 108
6.5 Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit JGroups mit 10 Agenten. 109
6.6 Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit Ersatz für JGroups mit 10 Agenten. 109
6.7 Einschwingverhalten aus Sicht der Bieter in der Simulation mit JGroups mit 50 Agenten. 110
6.8 Einschwingverhalten aus Sicht der Bieter in der Simulation mit Ersatz für JGroups mit 50 Agenten. 110
6.9 Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit JGroups mit 50 Agenten. 111
6.10 Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit Ersatz für JGroups mit 50 Agenten. 111
6.11 Einschwingverhalten aus Sicht der Bieter in der Simulation mit Ersatz für JGroups mit 100 Agenten. 112
6.12 Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit Ersatz für JGroups mit 100 Agenten. 112
6.13 Anzahl der Auktionen über die Zeit in der Simulation mit Ersatz für JGroups mit 100 Agenten. 113
6.14 Einschwingverhalten aus Sicht der Bieter in der Simulation mit Ersatz für JGroups mit 10 Agenten und mehreren zusammengesetzten Aktivitäten. . 113
6.15 Verhalten der Bieter im SLO-Spiels bei dynamischen Änderungen und 10 Agenten. 114

6.16 Verhalten der Bieter im SLO-Spiels bei dynamischen Änderungen und 100 Agenten. 115
6.17 Kommunikationsaufwand des SLO-Spiels bei dynamischen Änderungen und 10 Agenten. 115
6.18 Kommunikationsaufwand des SLO-Spiels bei dynamischen Änderungen und 100 Agenten. 116
6.19 Verhalten der Bieter im SLO-Spiels bei dynamischen Änderungen und 10 Agenten bei Überschreitung des Workflow SLAs. 116
6.20 Verhalten der Bieter im SLO-Spiels bei dynamischen Änderungen und 100 Agenten bei Überschreitung des Workflow SLAs. 117
6.21 Kommunikationsaufwand des SLO-Spiels bei dynamischen Änderungen und 10 Agenten bei Überschreitung des Workflow SLAs. 117
6.22 Kommunikationsaufwand des SLO-Spiels bei dynamischen Änderungen und 100 Agenten bei Überschreitung des Workflow SLAs. 118
6.23 Verhalten der Bieter des Workflows mit dem niedrigeren Wert. 119
6.24 Verhalten der Bieter des Workflows mit dem höheren Wert. 119
6.25 Anzahl durchgeführter Auktionen pro Zeit bei aktiviertem Scheduling. . . 120

Tabellenverzeichnis

5.1 Codestatistik in Programmzeilen pro Programmteil. 104

Quellcode-Verzeichnis

2.1 Beispielkonfiguration für einen JGroups-Protokoll-Stack 32
5.1 Das XML-Schema für Service-Definitionen 92
5.2 Das XML-Schema für Workflow-Definitionen 93
5.3 Das XML-Schema für Service-Knoten der verteilten Implementierung . . 94
5.4 Ausschnitt aus einem Trace-Log 96
5.5 Konfiguration von JGroups für TCP 99

Kapitel 1

Einführung

Unternehmen erweitern kontinuierlich ihre IT-Infrastruktur und verlassen sich in zunehmende Maße auf ihre Dienste. Je größer und komplexer die Strukturen werden und je größer die Abhängigkeit der Unternehmen von der IT-Infrastruktur wird, desto größer wird die Herausforderung die entstehenden Systeme zu verwalten und auch die Bedeutung des IT-Managements ansich.

Um der zunehmenden System-Komplexität gerecht zu werden und ein durchgängiges IT-Management über System- und Unternehmensgrenzen hinweg zu ermöglichen, wurde das Konzept des Service Level Managements (SLM) eingeführt (vgl. [Deb04]). Aus dem SLM geht ein sogenanntes Service Level Agreement (SLA) hervor, das ein Vertrag zwischen Anbieter eines Services und dessen Nutzer ist. Dieser Vertrag definiert für den angebotenen Service bestimmte Leistungskriterien in Form von Service Level Objectivs (SLO), an deren Einhaltung sich der Service-Anbieter bindet.

Um die Dienstgütekriterien eines SLAs einhalten zu können, muss auf Ereignisse in der IT-Infrastruktur durch geeignete Konfigurationsänderungen mittels eines Managementsystem auf die einzelnen Teilsysteme eingewirkt werden. Manuelles Management stößt hier durch die zunehmende Größe, Komplexität und erforderlichen kurzen Reaktionszeiten an Grenzen. Deshalb wird das manuelle Management zunehmend durch eine Automatisierung des Managements, sogenanntes Selbst-Management, abgelöst (vgl. [SK05]).

Durch die Einführung des Selbst-Managements wird jedes Teilsystem in die Lage versetzt selbsttätig Konfigurationsparameter zu verändern und sich dadurch an Änderungen in der Systemumgebung anzupassen.

Mit weiter zunehmender Komplexität der Systeme nimmt auch die Komplexität der Konfiguration des automatisierten Managementsystems immer weiter zu, weshalb man aktuell

die Entwicklung von selbstorganisierenden Systemen (vgl. [MWJ+07]) vorantreibt. Diese Systeme verändern nicht nur Konfigurationsparameter, sondern auch ihre eigentliche Konfiguration und die Struktur des Managementsystems.

Die meisten selbstorganisierenden Systeme sind durch bekannte Beispiele aus anderen Forschungsgebieten inspiriert. Ein sehr bekanntes Beispiel sind Ameisen-Algorithmen, welche durch die Beobachtung von Ameisen-Kolonien entstanden sind [DB05]. Ameisen kommunizieren indirekt mit Hilfe von Pheromonspuren, wodurch z.B. kürzeste Wege zwischen Futter und Nest gefunden werden. Ein weiteres Beispiel sind Mark-Mechanismen, welche aus den Wirtschaftswissenschaften stammen [FS02].

Der Ansatz dieser Mark-Mechanismen stammt aus einem Teilgebiet der Spieltheorie, dem algorithmischen Mechanismus-Design [NRTV07]. Der wesentliche Aspekt des algorithmischen Mechanismus-Design ist, dass sogenannte rationale Agenten modelliert werden. Rationale Agenten verhalten sich eigennützig, versuchen also stets nur den eigenen Nutzen zu maximieren.

Gleiches Verhalten ist auch zu beobachten, sobald das Management über administrative Domänengrenzen hinweg erfolgen soll. Jede Domäne versucht das Management üblicherweise so zu gestalten, dass ihr eigener Nutzen optimiert wird.

Eine parallel zur Entwicklung immer komplexerer Managementsysteme verlaufende Entwicklung, ist die zunehmende Verbreitung von Service-orientierten Systemen [Vas07]. Ein Kernbestandteil solcher Systeme sind Workflows, die bestimmte Geschäftsprozesse eines Unternehmens modellieren. Ein Workflow besteht aus verschiedenen Aktivitäten, die in einem gerichteten Graphen angeordnet sind.

Das Ziel der Arbeit ist die Entwicklung eines Mechanismus auf Basis des algorithmischen Mechanismus-Design für Service-orientierte Systeme. Dieser Mechanismus soll ein selbstorganisierender Mechanismus sein, der ein automatisiertes IT-Management durchführt. Der Mechanismus soll immer dann aktiv werden, wenn die im SLA definierten Leistungskriterien eines Services nicht eingehalten werden.

Die Arbeit beginnt mit einer Zusammenfassung der für das Verständnis notwendigen Grundlagen in Kapitel 2. Dort ist ein kurzer Überblick über Service Level Management, selbstorganisierende Systeme, das angesprochene algorithmische Mechanismus-Design und Gruppenkommunikation zu lesen.

In Abschnitt 3.1 wird zunächst die Problemstellung detailliert erläutert. Anschließend wird in den Abschnitten 3.2 und 3.3 für die Vereinbarung von Dienstgütekriterien für Service-orientierte Systeme die Definition eines solchen Service-orientierten Systems mit

den Eigenschaften aus dem SLM zusammengeführt. Für das beschriebene System wird anschließend analysiert welche prinzipiellen Management-Aktivitäten in einem solchen System durchgeführt werden können. Dabei ist das hauptsächliche Ziel immer die Einhaltung von Dienstgütekriterien gewährleisten zu können. Um eine systematische Analyse zu ermöglichen, wird das grundsätzliche Problem auf die Einhaltung der Antwortzeit eingeengt.

Anschließend werden die in Abschnitt 2.3 beschriebenen Grundlagen des algorithmischen Mechanismus-Design auf dieses System angewendet, um auf dieser Basis ein geeignetes selbstorganisierendes Managementverfahren zu entwickeln. Daraus entsteht das *SLO-Spiel* (Abschnitt 3.5), welches die Grundlage für die weitere Arbeit ist.

Das SLO-Spiel wird schrittweise entwickelt. Zuerst wird der zu Grunde liegende Mechanismus definiert (Abschnitt 3.5.1). Das SLO-Spiel ist prinzipiell eine verteilte Auktion, d.h. es muss festgelegt werden was in der Auktion versteigert wird, wer Bieter, wer Auktionator ist und wie sich Bieter und Auktionator verhalten.

In Abschnitt 3.5.2 wird darauf folgend eine vereinfachte Variante des SLO-Spiel theoretisch auf Basis der in Abschnitt 2.3 definierten Eigenschaften untersucht.

Die vereinfachte Variante des SLO-Spiels wird anschließend verallgemeinert (Abschnitt 3.5.4), so dass das entstehende verallgemeinerte SLO-Spiel alle in Abschnitt 3.4 definierten Lösungsaktivitäten umsetzt. Es wird analysiert, wie sich die theoretischen Eigenschaften durch die Verallgemeinerung verändern und mögliche dadurch entstehende Probleme aufgezeigt. Außerdem werden aus theoretischer Sicht noch offene generelle Probleme des SLO-Spiels beschrieben.

Nach der theoretischen Analyse der möglichen Lösungsaktivitäten und des SLO-Spiels soll das SLO-Spiel implementiert werden und anschließend eine Leistungsmessung und -Bewertung durchgeführt werden. Die für die Leistungsmessung und -Bewertung genutzte Methodik wird in Abschnitt 3.6 beschrieben.

Die folgenden Kapitel 4 und 5 beschäftigen sich mit der Umsetzung des SLO-Spiels. In Abschnitt 4.1 wird die grundsätzliche Architektur beschrieben. Es wird dabei sowohl eine Simulations- als auch eine verteilte Umgebung entwickelt. In beiden Umgebungen sollen Leistungsmessungen durchgeführt werden.

In Abschnitt 4.2.3 wird anschließend detailliert auf die Umsetzung des SLO-Spiels eingegangen. Im Wesentlich wird dabei der Ablauf des SLO-Spiels in einen Zustandsautomaten umgewandelt und die Kommunikationsstruktur zwischen einzelnen Agenten beschrieben.

Der Entwurf wird in Kapitel 5 um Aspekte der Implementierung ergänzt. Es wird be-

schrieben wie die Umgebungen konfiguriert werden (Abschnitt 5.1), wie Messdaten erhoben werden (Abschnitt 5.2), welche Kommunikationsprotokolle eingesetzt werden (Abschnitte 5.3 und 5.4) und wie die grafische Benutzeroberfläche aufgebaut ist (Abschnitt 5.5).

In Kapitel 6 wird schließlich die Umsetzung des SLO-Spiels ausgewertet. Es werden dazu verschiedene Messungen durchgeführt:

- Bewertung des Einschwingvorgangs
- Bewertung des Verhaltens bei dynamischen Lasten
- Verifikation einiger theoretischer Eigenschaften des verallgemeinerten SLO-Spiels und der vereinfachten Variante.

Abschließend wird in Kapitel 7 eine zusammenfassende Bewertung des SLO-Spiels gegeben. Dabei wird sowohl auf die theoretischen Ergebnisse, als auch die aus den Messungen ermittelten Leistungsdaten eingegangen. Zusätzlich wird beschrieben wie das SLO-Spiel in weiteren Arbeiten weiterentwickelt werden kann.

Kapitel 2

Grundlagen

2.1 Service Level Management

IT-Infrastrukturen in Unternehmen werden immer komplexer und schwieriger zu verwalten. Gleichzeitig steigt die Abhängigkeit der Firmen von ihrer IT-Infrastruktur immer weiter an. Im Lauf der Zeit wurden deshalb immer komplexere Management-Strukturen eingeführt, beispielsweise auf Basis der IT Infrastructure Library [Ogc01]. Das Management hat verschiedene Ziele, darunter die Einhaltung von Garantien bezüglich genutzter IT-Services oder die Reduktion der Kosten der IT-Infrastruktur.

Service-Level-Management (SLM) wurde in diesem Rahmen eingeführt, um IT-Services überwachen und gegebenenfalls korrigierend eingreifen zu können. Das Ziel von SLM ist dabei, bestimmte Garantien bezüglich der Dienstgüte (Quality of Service) geben zu können. Folgende Begriffe werden in der weiteren Arbeit genutzt. Die Begriffe basieren auf den Beschreibungen in [Deb04] und [LR99]:

- Ein *Workflow* ist ein Geschäftsprozess. Jeder Workflow hat eine bestimmte Struktur, bestehend aus *Aktivitäten* und Abhängigkeiten zwischen diesen Aktivitäten. Jede Aktivität wird durch einen Service ausgeführt.
- Ein *Service* ist eine Funktion oder eine Menge von Funktionen, die einem Service-Nutzer durch einen Service-Anbieter bereitgestellt wird. Es handelt es sich um bestimmte Software- oder Hardwarekomponenten, die zusammen den Service bilden.
- Die *Dienstgüte* beschreibt die Qualität, mit welcher der Service gegenüber dem Nutzer erbracht wird.

- Ein *Dienstgüteparameter* ist ein quantifizier- und messbarer Wert, der die Qualität eines Services beschreibt. Beispiele sind:
  - Die *Antwortzeit* ist die Zeitspanne, die zwischen dem Aufruf eines Services durch den Nutzer bis zur Beendigung dieses Aufrufs auf Seiten des Nutzers vergeht.
  - Der *Durchsatz* ist die Anzahl der verarbeiteten Aufrufe pro Zeiteinheit.
  - Die *Verfügbarkeit* ist die mittlere Wahrscheinlichkeit, die angibt, ob ein Service für einen Nutzer zu einem zufälligen Zeitpunkt tatsächlich nutzbar ist.
- Ein *Service-Level-Objective* (SLO) beschreibt eine Zielvorgabe für einen bestimmten Dienstgüteparameter.
- Ein *Service-Level-Agreement* (SLA) ist eine vertragliche Vereinbarung zwischen Nutzer und Anbieter eines oder mehrerer Services. In der Vereinbarung werden unter anderem alle beteiligten Services eindeutig identifiziert sowie Dienstgüteparameter und SLOs festgelegt.

2.2 Selbstorganisierende Systeme

Das Management von immer komplexer werdenden Systemen nimmt mit der Komplexität dieser Systeme zu. Um ein effektives SLM durchführen zu können, wird das Management selbst automatisiert, da manuelles Management mit der zunehmenden Komplexität nicht mehr schritthalten kann. *Selbst-Management* verändert Konfigurationsparameter einer Systemkomponente ohne die Struktur des Systems zu ändern.

Die Konfiguration des Managementsystems selbst wird ebenfalls immer komplexer. Deshalb ist ein weiterer Schritt auch die Konfiguration, also die Struktur des Managementsystems durch die einzelnen Komponenten selbst herstellen und modifizieren zu lassen. *Selbstorganisierende Systeme* werden also nicht mehr zentral konfiguriert, sondern finden ihre Struktur selbstständig und passen diese Struktur auch selbst weiter an.

Eine weitere interessante Problemstellung im Rahmen von selbstorganisierenden Systemen resultiert aus der Zusammenarbeit von Systemteilen, die unterschiedlichen administrativen Domänen zugeordnet sind, beispielsweise unterschiedliche Unternehmensteile oder Firmen. Jeder Systemteil hat dadurch lokale, eigennützige Ziele. Ein Teilsystem verhält sich also grundsätzlich so, dass es diese eigennützige Ziele anstrebt.

2.2.1 Klassifizierung

Eine genaue Einteilung der Systeme ist notwendig, um unterschiedliche Systemtypen und deren Eigenschaften untersuchen zu können. [MWJ+07] teilt Systeme in drei Klassen ein:

- *Adaptive Systeme* sind Systeme, die sich für bestimmte Eingaben akzeptabel verhalten.
- *Selbstverwaltende Systeme* sind adaptive Systeme, die sich für bestimmte Eingaben akzeptabel verhalten und sich für Eingaben, die ein unerwartetes Verhalten erzeugen, durch eine Kontrollfunktion sich selbst verändern und dadurch für diese Eingaben ebenfalls akzeptabel verhalten.
- *Selbstorganisierende Systeme* sind selbstverwaltende Systeme, welche die Kontrollfunktion dezentral implementieren und aus Komponenten bestehen, welche die Struktur des Systems verändern können, um sich anzupassen.

2.2.2 Autonomic Computing

Das von IBM entwickelte Konzept des Autonomic Computings [KC03] enthält eine weitere Klassifizierung von Systemen. Dabei werden vier *Selbst-** Eigenschaften unterschieden:

- *Selbst-Konfiguration* beschreibt eine automatische Konfiguration von Komponenten eines Systems anhand von vorgegebenen Richtlinien. Das System passt sich anhand dieser Richtlinien automatisch und selbständig an.
- *Selbst-Optimierung* beschreibt eine automatische, selbsttätige Optimierung des Systems durch die Komponenten des Systems. Jede Komponente versucht ihre eigene Performance zu verbessern.
- *Selbst-Heilung* beschreibt eine automatische Erkennung und Beseitigung von Fehlern im System.
- *Selbst-Schutz* beschreibt eine automatische Erkennung und Verteidigung gegen bösartige Angriffe oder einer Kette von Folgefehlern. Es wird durch Frühwarnsysteme der Zusammenbruch des Systems aufgehalten.

Vergleicht man die durch das Autonomic Computing definierten Selbst-*-Eigenschaften mit der zuvor dargestellten Klassifizierung. Die vorangegangene Klassifizierung unterscheidet verschiedene Arten der Anpassung. Autonomic Computing hingegen definiert Eigenschaften, die beschreiben, warum etwas angepasst werden soll. Die beiden Konzepte sind deshalb in gewisser Weise orthogonal zueinander. Beispielsweise muss ein selbstorganisierendes System nicht unbedingt Selbst-Heilung aufweisen, auf der anderen Seite ist ein sich selbst heilendes System nicht zwangsläufig auch ein dezentrales, sich selbst organisierendes System, kann es allerdings sein.

2.2.3 Beispiele

Viele Entwürfe zu selbstorganisierenden Systemen übertragen Konzepte aus anderen Forschungsgebieten in die Informatik. Beispielsweise gibt es selbstorganisierende Systeme in der Biologie oder den Wirtschaftswissenschaften:

- Ameisen-Kolonien, Bienen- oder Wespenschwärme
- Markt-Mechanismen

Ant Colony Optimization

Ein bekanntes Beispiel selbstorganisierender Systeme sind sogenannte Ameisen-Algorithmen. Die Algorithmen stammen aus Beobachtungen über Ameisen-Kolonien in der Biologie und wurden zuerst durch [DG97] eingeführt.

Ein beobachtetes Verhalten von Ameisen wird im folgenden beschrieben: Ameisen wandern zuerst zufällig umher, bis sie Futter gefunden haben, dort evaluieren sie die Menge und Qualität des Futters und nehmen anschließend einen Teil davon mit zur Kolonie. Auf dem Rückweg wird je nach Qualität und Quantität des Futters eine unterschiedlich ausgeprägte Pheromonspur hinterlassen. Je stärker die Pheromonspur, desto mehr Ameisen bewegen sich anschließend auf diesem Pfad. Durch die indirekte Kommunikation über die Pheromonspuren können Ameisen das Problem lösen, kürzeste Pfade zwischen dem Futter und der Kolonie zu finden [DAGP90].

Dieses Verhalten lässt sich auf künstliche Ameisen-Kolonien übertragen, z.B. um Routing-Probleme oder andere kombinatorische Optimierungsprobleme zu lösen [GTA99]. Weiterhin lässt sich ein Ameisen-Algorithmus auch dezentral implementieren, so dass das entstehende System zur Klasse der selbstorganisierenden Systeme gehört [RKKC05].

[DB05] gibt einen guten Überblick über Eigenschaften, Anwendungen und noch offene Probleme zu diesem Themengebiet.

Markt-Mechanismen

Markt-Mechanismen werden mittels der Spieltheorie untersucht. Ein Teil der Spieltheorie ist das Gebiet des Mechanismus-Design welches Markt-Mechanismen mit geeigneten Eigenschaften entwickelt [FT91]. Mechanismus-Design entwickelt spezielle Mechanismen für Situationen, in denen mehrere eigennützige Agenten an einem Problem arbeiten, beispielsweise die Zuteilung von Ressourcen. Der entwickelte Mechanismus hat das Ziel, ein global möglichst gutes Ergebnis zu erreichen, während die Agenten selbst versuchen, ihren eigenen lokalen Nutzen zu erhöhen.

Das speziellere Forschungsgebiet des verteilten algorithmischen Mechanismus-Design [FS02] entwickelt Mechanismen, die letztendlich selbstorganisierende Systeme sind. Ein verteilter Mechanismus in diesem Sinne ist:

- adaptiv, da er sich für unterschiedliche Eingaben akzeptabel verhält.
- selbstverwaltend, da er ohne externe Eingriffe Änderungen am System durchführt.
- selbstorganisierend, da Agenten nur dann mit anderen Agenten kommunizieren, wenn es für sie selbst von Vorteil ist. Machen es die Eingaben notwendig, verändern die Agenten eigennützig die Struktur des Systems.

Der Ansatz des Mechanismus-Design wird im weiteren Verlauf dargestellt und auf die Problemstellung der Arbeit angewandt.

2.3 Algorithmisches Mechanismus-Design

Mechanismus-Design ist ein Teilgebiet der Spieltheorie und stammt damit ursprünglich aus den Wirtschaftswissenschaften. Die klassische Spieltheorie untersucht die Interaktion von nicht-kooperativen Multiagenten-Systemen [MVN44]. Mechanismus-Design [FT91, MCWG95] geht den umgekehrten Weg, d.h. es werden gewünschte Eigenschaften definiert und anschließend ein Mechanismus gesucht, der diese Eigenschaften erfüllt.

Klassisches Mechanismus-Design betrachtet dabei die für die Informatik wichtigen algorithmischen Eigenschaften nicht. Diese Eigenschaften spielen dagegen im algorithmischen Mechanismus-Design eine zentrale Rolle. Entstanden ist das Forschungsgebiet des

algorithmischen Mechanismus-Designs im Wesentlichen durch das Aufkommen des Internets. Beispiele dafür sind P2P-Anwendungen wie Bittorrent [JA05] oder Interdomain-Routing [FPSS05], in denen nicht-kooperative Agenten (Peers, Router) ein gemeinsames Problem bearbeiten, dabei allerdings nur ihre individuelle Position verbessern wollen (Download-Rate, effektive Nutzung der Netzwerk-Verbindungen).

2.3.1 Spieltheorie

Die Spieltheorie untersucht Spiele mit mehreren Teilnehmern, genannt Agenten. Spiele sind dabei beliebige Interaktionen zwischen diesen Agenten. Eine Grundannahme ist dabei, dass die Teilnehmer sich rational und eigennützig verhalten, also immer danach streben, ihren eigenen Nutzen zu maximieren.

Am Beispiel des *Gefangenendilemma* [NRTV07] lassen sich einige der wichtigsten Konzepte informell darstellen. Anschließend wird ein Modell formal definiert, das beim algorithmischen Mechanismus-Design Verwendung findet.

Beispiel 2.3.1 (Gefangenendilemma)**.** Zwei Gefangene werden jeweils unabhängig vor die Wahl gestellt, auszusagen oder zu schweigen. Schweigen beide, ist die Strafe für beide 2 Jahre Haft. Schweigt einer und der andere sagt aus, bekommt der Schweigende 5 Jahre Haft und der Aussagende 1 Jahr Haft. Sagen beide aus, bekommen beide eine Haftstrafe von 4 Jahren. Bis beide eine Entscheidung getroffen haben, dürfen beide Gefangenen nicht miteinander reden. Die resultierenden Haftstrafen kennen beide.

Jeder Gefangene wählt seine *Strategie* unabhängig vom anderen. Jeder hat dabei die Wahl zwischen den zwei Strategien Aussagen oder Schweigen. Die Kosten in Form der zu erwartenden Strafe lassen sich folgendermaßen darstellen:

Gefangener 2 \ Gefangener 1	Schweigen	Aussagen
Schweigen	2 | 2	5 | 1
Aussagen	1 | 5	4 | 4

Abbildung 2.1: Die Darstellung des Gefangenendilemmas in Matrix-Form.

Die Analyse des Problems ergibt, dass es nur eine stabile Strategie für beide Gefangenen gibt. Eine *stabile Strategie* ist in der Spieltheorie eine Strategie die den Nutzen eines Agenten lokal maximiert, d.h. der Agent kann durch Abweichung von dieser Strategie seinen Nutzen nicht selbst weiter erhöhen.

Im Fall des Gefangenendilemmas ist eine stabile Strategie, dass beide Gefangenen aussagen. Tun dies beide Gefangenen, kann keiner durch eine Abweichung von dieser Strategie seine eigene Situation verbessern. Auffallend ist dabei, dass es sich nicht um ein globales Optimum handelt. Die entstandene *Gleichgewichtsstrategie* wird nach John Nash auch *Nash-Gleichgewicht* genannt [Nas50, Nas51]. Das globale Optimum, beide schweigen, ist hingegen nicht stabil. Jeder der beiden Gefangenen kann seine Situation individuell verbessern, gleichzeitig wird dadurch aber die Situation des anderen verschlechtert.

2.3.2 Einmalige und wiederholte Spiele

Die Interaktion der Agenten kann prinzipiell einmalig oder wiederholt stattfinden. Grundsätzlich lassen sich drei Arten unterscheiden [FT91]:

- *einmalige* Spiele
- *endlich oft wiederholte* Spiele
- *unendlich oft wiederholte* Spiele

Wiederholte Spiele können zusätzlich zu den Gleichgewichtsstrategien aus einem einmaligen Spielen weitere Gleichgewichtsstrategien besitzen. Im unendlich oft wiederholten Fall lassen sich alle Ergebnisse erreichen, die besser als ein Nash-Gleichgewicht sind. Dies folgt aus den *Folk-Theoremen*. Jedes dieser Folk-Theoreme basiert auf dem gleichen Prinzip. Es wird solange eine bestimmte Strategie gespielt, bis ein Agent davon abweicht. Die anderen Agenten bestrafen nun den abweichenden Agenten durch eine Strategieänderung, so dass die Abweichung langfristig zu einem schlechteren Ergebnis führt.

Die meisten Folk-Theoreme setzen vollständiges Wissen wie im Gefangenendilemma voraus [FT91, Fri71, AS92], um eine sogenannte Minimax-Strategie als Bestrafung spielen zu können. Die Minimax-Strategie minimiert den maximalen Nutzen von anderen Agenten [NRTV07]. Im Gefangenendilemma ist dies gleichzeitig eine Strategie die zu einem Nash-Gleichgewicht im nur einmal durchgeführten Spiel führt. Auch für den Fall von unvollständigem Wissen wurde in [FLM94] eine Variante des Folk-Theorems bewiesen.

Am Beispiel des *wiederholten Gefangenendilemma* lässt sich die Existenz weiterer Gleichgewichts-Strategien zeigen:

Beispiel 2.3.2 (wiederholtes Gefangenendilemma)**.** Im wiederholten Gefangenendilemma gibt es mehrere Nash-Gleichgewichte. Neben dem Nash-Gleichgewicht aus dem einmaligen Spiel, d.h. immer auszusagen, besteht weiterhin die Möglichkeit immer zu schweigen. Sobald einer der beiden Gefangenen davon abweicht, um seine eigene Situation zu verbessern, wird der andere Gefangene in Zukunft immer die Minimax-Strategie wählen, also aussagen, wodurch der kurzzeitige Gewinn des abweichenden Gefangenen durch die Bestrafung aufgezehrt wird.

Da die Abweichung für einen Gefangenen langfristig im unendlich oft wiederholten Fall ein schlechteres Ergebnis bringt, werden beide Gefangenen immer schweigen:

$$\text{Strafe} = \sum_{i=1}^{\infty} 2$$

Weicht ein Gefangener in der j-ten Wiederholungen ab, wird danach nur noch die Gleichgewichtsstrategie des einmaligen Spiels gespielt, dies ist offensichtlich schlechter:

$$\text{Strafe} = \sum_{i=1}^{j-1} 2 + 1 + \sum_{k=j+1}^{\infty} 4$$

Ist dagegen bekannt, wie oft das Spiel durchgeführt wird, ändert sich die Strategie in der letzten Wiederholung. In der letzten Wiederholung würde ein einzelner Gefangener seine Situation durch einseitiges Abweichen verbessern und der andere Gefangene hätte keine Möglichkeit mehr die Bestrafung durchzuführen. Deshalb weichen beide Agenten auf das Gleichgewicht des einmaligen Spiels aus. Frühere Abweichungen werden dagegen identisch wie beim unendlich oft wiederholten Spiel bestraft. Bei n Wiederholungen ist das Ergebnis der Gleichgewichtsstrategie:

$$\text{Strafe} = \sum_{i=1}^{n-1} 2 + 4$$

2.3.3 Agenten-Typen

Generell werden durch die Spieltheorie und das Mechanismus-Design nur Agenten betrachtet, die sich eigennützig und rational verhalten. In vielen Fällen ergibt es allerdings

auch Sinn, andere Typen von Agenten zu betrachten [SP03, FS02, DJP03]. Mögliche Agenten-Typen sind:

- *Gehorsame* Agenten, die sich exakt an die Spezifikation des Entwicklers halten. Solche Agenten sind in der Informatik der Normalfall.
- *Fehlerhafte* Agenten sind alle Arten von Agenten, die sich auf Grund von Defekten nicht an die Spezifikation halten können.
- *Feindliche* Agenten agieren gegen andere Agenten. Beispielsweise führt ein solcher Agent kryptographische Angriffe durch.
- *Rationale* Agenten sind die typischen Agenten der Spieltheorie. Sie verhalten sich strategisch, rational und zum eigenen Vorteil.
- *Irrationale* Agenten verhalten sich ebenfalls strategisch und zum eigenen Vorteil. Allerdings liegt ihrem Verhalten ein anderes Nutzenmodell zu Grunde als den rationalen Agenten. Sie verhalten sich dem Mechanismus gegenüber irrational, d.h. das Verhalten eines Agenten ist durch den Mechanismus nicht vorhersehbar.

Mechanismus-Design betrachtet das Zusammenspiel rationaler Agenten. In einigen Fällen ist es allerdings bisher notwendig, Agenten einzusetzen, die nicht rational sind, sondern ein vorgegebenes Protokoll exakt ausführen. Der auf BGP basierende Mechanismus in [FPSS05] benötigt beispielsweise eine zentrale Bank, um Zahlungen der einzelnen Agenten zu verwalten. In diesem zentralen Mechanismus werden die Agenten für die Dienstleistung der Datenübertragung bezahlt, d.h. je höher die genutzte Bandbreite desto höher ist die Bezahlung.

Aus Fehlertoleranz- und Sicherheitsgründen sollte eine reale Implementierung eines Mechanismus auch fehlerhafte, respektive feindliche Knoten mit betrachten. Das Verhalten irrationaler Agenten kann ebenso das Ergebnis eines Mechanismus verschlechtern. Beispielsweise haben *antisoziale* Agenten das Ziel, anderen Agenten zu schaden [Bra00, BW01].

In der weiteren Arbeit werden rationale Agenten betrachtet, die sich an das durch den Mechanismus definierte Nutzenmodell halten. Ausnahmen davon werden gesondert hervorgehoben. Ein weiteres wichtiges Konzept besteht darin, dass die rationalen Agenten normalerweise Risiko-neutral sind, d.h. wenn ein Agent durch eine Strategie mit einer Wahrscheinlichkeit von 0.5 einen Nutzen von 2 hat, ist die Strategie gleichwertig mit einer anderen Strategie, die mit einer Wahrscheinlichkeit von 1 einen Nutzen von 1 für den Agenten hat.

2.3.4 Modell

Im folgenden wird das in der Arbeit verwendete Modell definiert. Das dem Mechanismus-Design zugrunde liegende Modell wird unter anderem in [NR01, NRTV07, Ste08] detailliert beschrieben.

Im weiteren Verlauf werden folgende Schreibweisen benutzt:

- Ein Vektor $v = (v_1, \cdots, v_n)$ enthält für jeden Agenten genau einen Eintrag.
- Ein Vektor $v_{-i} = (v_1, \cdots, v_{i-1}, v_{i+1}, \cdots, v_n)$ enthält für alle Agenten einen Eintrag, ausgenommen für den Agenten i.
- Der Vektor v kann auch als $v = (v_i, v_{-i})$ geschrieben werden.

Definition 2.3.3 (Mechanismus-Design Problem)**.** Ein Mechanismus-Design Problem wird durch ein gewünschtes Ergebnis beschrieben, das von einer Anzahl rationaler Agenten erreicht werden soll. Dabei gelten folgende Annahmen:

- Ein *statischer Mechanismus* ist ein Tupel $M =_{def} (A, \mathfrak{S}, O, f, p_1, \cdots, p_n)$, mit einer endlichen Menge *Agenten* $A \subset \mathbb{N}$, einer Menge möglicher *Strategien* $\mathfrak{S}$, einer Menge möglicher *Ergebnisse* O, der *Ergebnisfunktion* $f : \mathfrak{S}^n \rightarrow O$ und *Zahlungsfunktionen* $p_i : \mathfrak{S}^n \rightarrow \mathbb{R}$ für alle Agenten $i \in A$.
- Jeder Agent i besitzt private Informationen, die im sogenannten *Typ* $t_i \in T_i$ widergespiegelt werden. Die Menge T_i ist dabei für den Agenten i eine Teilmenge aller möglichen Typen $\mathfrak{T}$.
- Jeder Agent i hat mehrere mögliche Strategien $S_i \subseteq \mathfrak{S}$. Eine Strategie ist definiert als eine Funktion $s_i : T \times \mathfrak{S}^{n-1} \rightarrow S_i$. Anschaulich ist die Strategie des Agenten i vom eigenen Typ und von den Strategien aller anderen Agenten abhängig.
- Für jeden Strategievektor $s = (s_1, \cdots, s_n)$ existiert ein Ergebnis $o \in O$, das durch die Ergebnisfunktion f berechnet wird.
- Die Ergebnisfunktion wird auch *soziale Entscheidungsfunktion* genannt. Eine soziale Entscheidungsfunktion ist dann *effizient*, wenn sie ein bestimmtes Kriterium maximiert.
- Für jeden Agenten hat ein Ergebnis einen bestimmten Wert, der durch eine *Wertfunktion* $v_i : T_i \times O \rightarrow \mathbb{R}$ beschrieben wird. Der Mechanismus teilt jedem Agenten i

eine Bezahlung mit der Bezahlungsfunktion p_i in einer gemeinsamen Währung zu. Der tatsächliche Nutzen für einen Agenten i wird durch die *lineare Nutzenfunktion* u_i beschrieben:

$$u_i(o, p_i(s_i, s_{-i}), t_i) =_{def} v_i(t_i, o) - p_i(s_i, s_{-i})$$

Ein rationaler Agent versucht, diese Nutzenfunktion zu maximieren.

Definition 2.3.4 (wiederholter Mechanismus)**.** Ein *wiederholter Mechanismus* besteht aus der wiederholten Ausführung eines statischen Mechanismus. Jede Ausführung des statische Mechanismus ist ein *Teilspiel* k.

Die Nutzenfunktion $\mathfrak{u}_i$ eines Agenten i in einem wiederholten Mechanismus ist folgendermaßen definiert, wobei $u_{i,k}$ die Nutzenfunktion aus dem Teilspiel k ist [AS92, FT91]:

$$\mathfrak{u}_i =_{def} (1 - \delta) \sum_{k=1}^{\infty} \delta^k u_{i,k},$$

wobei δ ein Diskontfator $0 < \delta < 1$ ist. In einem wiederholten Mechanismus versucht ein Agent diese Nutzenfunktion zu maximieren.

Durch den Diskontfaktor δ wird der zukünftige Nutzen geringer gewichtet. $(1 - \delta)$ ist ein Normalisierungsfaktor. Intuitiv kann man den Diskontfaktor als Maß der Geduld eines Agenten betrachten. Bei $\delta \to 0$ ist der Agent vollständig ungeduldig, dagegen ist ein Agent bei $\delta \to 1$ vollständig geduldig. Ist ein Agent ungeduldig ist der aktuelle Nutzen wichtiger als Nutzen in der Zukunft. Je geduldiger ein Agent ist, desto höher wird der Nutzen der Zukunft gewichtet und desto eher wird ein Agent warten, um später einen höheren Nutzen zu erzielen als in der Gegenwart.

2.3.5 Eigenschaften eines Mechanismus

Die wichtigsten Eigenschaften eines Mechanismus werden hier definiert. Einen weitergehenden Überblick über die Eigenschaften von Mechanismen geben [DJP03, NR01, NRTV07, Ste08].

Definition 2.3.5 (Prinzip des direkten Enthüllens)**.** Ein Mechanismus $M = (A, \mathfrak{S}, O, f, p_1, \cdots, p_n)$ wird *direkt enthüllend* genannt, wenn für alle Agenten $i \in A$, $S_i = T_i$ gilt, d.h. für jeden Agent i ist die einzige mögliche Strategie einen Typ $t_i \in T_i$ zu deklarieren. Der deklarierte Typ muss dabei nicht den tatsächlichen privaten Informationen entsprechen.

Definition 2.3.6 (Individuelle Rationalität)**.** Ein Mechanismus $M = (A, \mathfrak{S}, O, f, p_1, \cdots, p_n)$ ist *individuell-rational*, wenn ein Agent durch die Teilnahme am Mechanismus seinen Nutzen im Gegensatz zur Nichtteilnahme erhöhen oder mindestens beibehalten kann. Es gilt also für jeden Agent i, dem Nutzen bei Nichtteilnahme $\bar{u}(t_i)$ und für jeden Strategievektor $(s_i, s_{-i}) \in \mathfrak{S}^n$:

$$u_i(f(s_i, s_{-i}), p_i(s_i, s_{-i}), t_i) \geq \bar{u}(t_i)$$

Definition 2.3.7 (Anreizkompatibilität)**.** Ein Mechanismus $M = (A, \mathfrak{S}, O, f, p_1, \cdots, p_n)$ ist *anreizkompatibel*, wenn die Anreize des Mechanismus (in Form von Zahlungen) für jeden Agenten i ihn dazu bringen, wahrheitsgemäße Deklarationen gegenüber dem Mechanismus zu machen. Der Mechanismus M ist anreizkompatibel, wenn für jeden Agenten $i \in A$, der wahrheitsgemäßen Strategie $s_i \in \mathfrak{S}$ und jeder anderen Strategie $s_i' \in \mathfrak{S}$ gilt:

$$u_i(f(s_i, s_{-i}), p_i(s_i, s_{-i}), t_i) \geq u_i(f(s_i', s_{-i}), p_i(s_i', s_{-i}), t_i)$$

Anreizkompatibilität wird in der Literatur oft auch als strategische Robustheit oder Wahrhaftigkeit bezeichnet.

Definition 2.3.8 (Pareto-Optimalität)**.** Die soziale Entscheidungsfunktion f eines Mechanismus M ist *pareto-optimal*, wenn das gewählte Ergebnis $o \in O$ den Wert eines Ergebnisses für alle Agenten $i \in A$ maximiert, so dass es kein anderes Ergebnis $o' \in O$ gibt, das für einen Agenten den Wert steigert, für alle anderen Agenten der Wert mindestens gleich bleibt. Es gilt also für das Ergebnis $o \in O$:

$$v_i(o, t_i) \geq v_i(o', t_i)$$

Ein Ergebnis $o \in O$ wird von einem anderen Ergebnis $o' \in O$ *pareto-dominiert*, wenn es für einen Agenten den Wert erhöht und für alle anderen Agenten der Wert gleich bleibt oder ebenfalls erhöht wird.

Definition 2.3.9 (Budget-Ausgeglichenheit)**.** Ein Mechanismus M ist *budget-ausgeglichen*, wenn die Summe der Zahlungen p_i, aller Agenten $i \in A$, 0 beträgt:

$$\sum_{i=1}^{n} p_i(s_i, s_{-i}) = 0$$

2.3.6 Gleichgewichts-Strategien

Wie schon beim Beispiel mit dem Gefangenendilemma gezeigt, ist die prinzipielle Lösung eines Spiels ein Gleichgewicht. Hierfür gibt es verschiedene Modelle, die unterschiedliche Eigenschaften haben. Das allgemeinste Gleichgewicht ist das sogenannte Nash-Gleichgewicht [Nas50, Nas51].

Ausgehend vom Nash-Gleichgewicht kann man die Eigenschaften verschiedener anderer Gleichgewichtsstrategie einengen. Jedes der anderen Gleichgewichts-Konzepte ist gleichzeitig auch ein Nash-Gleichgewicht. Eine gute Einführung und Übersicht zu Gleichgewichtskonzepte geben [FT91, NRTV07].

Definition 2.3.10 (Nash-Gleichgewicht). Ein Strategievektor (s_i, s_{-i}) befindet sich in einem *Nash-Gleichgewicht*, wenn für jeden Agent i, die gewählte Strategie s_i, jede andere Strategie $s'_i \in \mathfrak{S}$ und die gewählten Strategien der Mitspieler $s_{-i} \in \mathfrak{S}^{n-1}$ gilt:

$$u_i(f(s_i, s_{-i}), p_i(s_i, s_{-i}), t_i) \geq u_i(f(s'_i, s_{-i}), p_i(s'_i, s_{-i}), t_i)$$

Spielen die Agenten ein Nash-Gleichgewicht, kann also kein Agent selbst seinen Nutzen durch Abweichen von dieser Strategie verbessern. Um im allgemeinen Fall ein Nash-Gleichgewicht zu erreichen, muss jeder einzelne Agent die Strategien aller anderen Agenten kennen, da es in einem Spiel mehrere solcher Gleichgewichte geben kann. Weiterhin ist die Berechnung eines Nash-Gleichgewichts ein vermutlich schweres Problem [CS03, DGP06], wodurch es als allgemeines Lösungskonzept in der Informatik nur eingeschränkt brauchbar ist.

Definition 2.3.11 (Bayesian-Nash-Gleichgewicht). Ein Strategievektor (s_i, s_{-i}) befindet sich in einem *Bayesian-Nash-Gleichgewicht*, wenn für jeden Agent $i \in A$, die gewählte Strategie $s_i \in \mathfrak{S}$ und jede andere Strategie $s'_i \in \mathfrak{S}$ gilt:

$$E_{D_{-i}}[u_i(f(s_i, s_{-i}), p_i(s_i, s_{-i}), t_i)] \geq E_{D_{-i}}[u_i(f(s'_i, s_{-i}), p_i(s'_i, s_{-i}), t_i)]$$

Dabei ist $E_{D_{-i}}[]$ der Erwartungswert für die Wahl der Strategien $s_{-i} \in \mathfrak{S}^{n-1}$ über der Verteilung D_{-i}.

Während man für den allgemeinen Fall beim Nash-Gleichgewicht vollständige Informationen über alle Agenten benötigt, um die eigenen Strategie festzulegen, wird dies im Bayesian-Nash-Gleichgewicht durch den Gedanken von wahrscheinlichen Strategien der

anderen Agent ergänzt. Jeder Agent muss folglich nur noch eine Wahrscheinlichkeitsverteilung kennen.

Definition 2.3.12 (Ex-post-Nash-Gleichgewicht)**.** Ein Strategievektor (s_i^*, s_{-i}^*) befindet sich in einem *Ex-post-Nash-Gleichgewicht*, wenn für jeden Agenten $i \in A$, die gewählte Strategie $s_i^* \in \mathfrak{S}$, jede andere Strategie $s_i' \in \mathfrak{S}$ und jeden Typ $t_i \in T_i$ und alle möglichen Typen der anderen Agenten $t_{-i} \in \mathfrak{T}^{n-1}$ gilt:

$$u_i(f(s_i^*(t_i), s_{-i}^*(t_{-i})), p_i(s_i^*(t_i), s_{-i}^*(t_{-i})), t_i) \geq$$
$$u_i(f(s_i'(t_i), s_{-i}^*(t_{-i})), p_i(s_i'(t_i), s_{-i}^*(t_{-i})), t_i)$$

Das Ex-post-Nash-Gleichgewicht bedeutet, dass jeder Agent unabhängig von den anderen Agenten seine Gleichgewichtsstrategie wählen kann. Die Strategiewahl jedes Agenten ist im besonderen unabhängig von den Typen aller Agenten, nicht aber von der tatsächlichen Strategiewahl der anderen Agenten. Da diese aber auch immer ihre Ex-post-Nash-Strategie wählen werden, wird das Gleichgewicht gespielt. Ex-post bedeutet hier, dass ein Agent auch nach dem Bekanntwerden der Strategien aller anderer Agenten seine Strategie nicht mehr ändern will.

Definition 2.3.13 (Dominantes Gleichgewicht)**.** Ein *dominantes Gleichgewicht* besteht dann, wenn für einen Strategievektor (s_i, s_{-i}), für alle Agenten $i \in A$, der gewählten Strategie $s_i \in \mathfrak{S}$, jeder anderen Strategie $s_i' \in \mathfrak{S}$ und aller möglichen Strategien $s_{-i} \in \mathfrak{S}^{n-1}$ der anderen Agenten gilt:

$$u_i(f(s_i, s_{-i}), p_i(s_i, s_{-i}), t_i) \geq u_i(f(s_i', s_{-i}), p_i(s_i, s_{-i}), t_i)$$

Die wichtigste Eigenschaft des dominanten Gleichgewichts und gleichzeitig die wichtigste Unterscheidung zum Ex-post-Nash-Gleichgewicht ist, dass jeder Agent immer seinen Nutzen maximieren kann, unabhängig davon, ob die anderen Agenten sich ebenfalls rational verhalten. Das Ex-post-Nash-Gleichgewicht und das dominante Gleichgewicht sind jeweils anreizkompatibel (siehe Definition 2.3.7), sofern die wahrheitsgemäße Strategie mit der Gleichgewichtsstrategie übereinstimmt.

2.3.7 Komplexität

Die Komplexität eines Mechanismus ist ein wichtiges Maß, ob ein Mechanismus praktikabel implementiert werden kann. In [Par01] und [Ste08] wird ein Mechanismus in fol-

genden vier Teilproblemen betrachtet:

- *Komplexität der Typbestimmung*: Aus den privaten Informationen muss ein Agent zuerst seinen Typ selbst bestimmen.
- *Komplexität der Strategieberechnung*: Ist das angewandte Gleichgewichtskonzept kein dominantes oder Ex-post-Nash-Gleichgewicht, müssen die Strategien der anderen Teilnehmer modelliert und berechnet werden, um eine eigene Strategie wählen zu können.
- *Komplexität der Ergebnisbestimmung*: Haben die Agenten ihre Strategieentscheidung an den Mechanismus übermittelt, muss das Ergebnis berechnet werden, z.B. ist bei kombinatorischen Auktionen die Ergebnisbestimmung im Allgemeinen ein NP-vollständiges Problem.
- *Komplexität der Kommunikation*: Die Anzahl der auszutauschenden Nachrichten zwischen allen beteiligten Agenten, um ein Ergebnis des Mechanismus zu berechnen.

Ein Mechanismus ist dann in polynomieller Zeit berechenbar, wenn alle Teilprobleme in polynomieller Zeit berechenbar sind und polynomiell viele Kommunikationsschritte benötigt werden.

2.3.8 Vickrey-Clarke-Groves Mechanismen

Eine bekannte und wichtige Gruppe von Mechanismen mit interessanten Eigenschaften sind die Vickrey-Clarke-Groves-Mechanismen (VCG-Mechanismen) [NRTV07].

Definition 2.3.14 (Vickrey-Clarke-Groves Mechanismus)**.** Ein Mechanismus $M = (A, \mathfrak{S}, O, f, p_1, \cdots, p_n)$ wird als Vickrey-Clarke-Groves-Mechanismus bezeichnet, wenn gilt

- $f(s_1, \cdots, s_n) \in \max_{o \in O} \sum_{i=1}^{n} v_i(o, t_i)$, d.h. f maximiert das soziale Wohl und
- es Funktionen $h_1, \cdots, h_n$ gibt, für die gilt $h_i : \mathfrak{S}^{n-1} \to \mathbb{R}$. Für alle p_i gilt dann: $p_i(s_1, \cdots, s_n) = h_i(s_{-i}) - \sum_{j \neq i} v_j(f(s_1, \cdots, s_n), t_j)$.

Satz 2.3.15. *Jeder VCG-Mechanismus ist anreizkompatibel.*

Beweis. Beweis siehe [NRTV07]. □

Wenn die Funktionen h_i entsprechend gewählt werden, ist ein VCG-Mechanismus individuell rational. Ein Beispiel dafür ist die Clarke-Pivot-Regel [NRTV07].

Definition 2.3.16 (Clarke-Pivot-Regel)**.** Die Wahl $h_i(v_{-i}) = \max_{b \in O} \sum_{j \neq i} v_j(b, t_j)$ wird als Clarke-Pivot-Bezahlung bezeichnet. Die Bezahlung für Agent i ist mit dieser Regel $p_i(s_1, \cdots, s_n) = \max_{b \in O} \sum_{j \neq i} v_j(b, t_j) - \sum_{j \neq i} v_j(o, t_j)$, wobei $o = f(s_1, \cdots, s_n)$.

Satz 2.3.17. *Ein VCG-Mechanismus mit Clarke-Pivot-Bezahlungen ist individuell rational.*

Beweis. Beweis siehe [NRTV07]. □

Anschaulich bezahlt jeder Agent durch die Clarke-Pivot-Regel den Schaden, den er durch seine eigene Strategie den anderen Agenten zufügt.

VCG-Mechanismen sind deswegen so interessant, weil sie in einmaligen Spielen zu einem dominanten Gleichgewicht führen. Für dieses Gleichgewicht muss ein Agent keinerlei Wissen über die anderen Agenten besitzen. Sie eignen sich deshalb auch gut für verteilte Mechanismen, da sich die Kommunikation zwischen den Agenten auf die Deklaration des Typ gegenüber dem Mechanismus und die Mitteilung des Ergebnisses beschränkt. VCG-Mechanismen sind im Allgemeinen allerdings nicht budget-ausgeglichen [KP97].

2.3.9 Auktionen

Definition 2.3.18. Eine *Auktion* ist ein Mechanismus $M = (A, \mathfrak{S}, O, f, p_1, \cdots, p_n)$, dessen Ergebnismenge O aus Zuordnungen einer Menge von Objekten $\mathfrak{O}$ zu der Menge der Agenten A besteht. Jeder Agent bezahlt für das Objekt, das er zugeordnet bekommt, einen Betrag in einer gemeinsamen Währung.

Es gibt verschiedene Arten von Auktionen:

- Die englische Auktion ist die bekannteste Auktionsform. Hier wird der zu zahlende Betrag schrittweise durch Gebote von einzelnen Agenten erhöht.
- Bei der holländischen Auktion wird der zu zahlende Betrag schrittweise von einem Anfangsbetrag reduziert, bis ein Agent den momentanen Betrag bietet.

- Bei der Vickrey-Auktion gibt jeder Agent ein geheimes Gebot ab. Das höchste Gebot gewinnt, der Gewinner muss allerdings nur den Betrag des zweithöchsten Gebots bezahlen.

Satz 2.3.19 (Prinzip des äquivalenten Ertrags)**.** *Ohne Manipulation und mit privaten Wertfunktionen erzielen die englische, die holländische und die Vickrey-Auktion äquivalente Erträge.*

Beweis. Beweis siehe [Vic61]. □

Da die Vickrey-Auktion den niedrigsten Kommunikationsaufwand zwischen Bieter und Auktionator benötigt, wird im weiteren Verlauf der Arbeit diese Auktionsform genutzt.

Die nötige Kommunikation bei der Vickrey-Auktion besteht aus der Bekanntmachung der Auktion, der Abgabe des Gebots von jedem Bieter und der Benachrichtigung des Gewinners der Auktion. Bei der englischen und der holländischen Auktion müssen dagegen pro Schritt alle Bieter benachrichtigt werden.

Satz 2.3.20 (Vickrey-Auktion)**.** *Die Vickrey-Auktion, in der nur ein einziges Objekt versteigert wird, ist ein VCG-Mechanismus mit der Clarke-Pivot-Regel.*

Der Satz lässt sich direkt durch den hier vom Autor erbrachten Beweis konstruieren.

Beweis. Es gibt zwei Fälle für einen Agenten i und das Ergebnis o:

- Der Agent i hat den Zuschlag für das Objekt nicht bekommen, dann gilt

$$\max_{b \in O} \sum_{j \neq i} v_j(b, t_j) = \sum_{j \neq i} v_j(o, t_j)$$

  und folglich

$$p_i(s_1, \cdots, s_n) = 0.$$

- Der Agent i hat den Zuschlag für das Objekt bekommen, dann gilt

$$\max_{b \in O} \sum_{j \neq i} v_j(b, t_j) = \max_{b \in O} v_{k \neq i}(b, t_k).$$

  Agent k ist der Agent mit dem zweithöchsten Gebot. Außerdem gilt

$$\sum_{j \neq i} v_j(o, t_j) = 0.$$

Die Bezahlung des Agenten i ist folglich

$$p_i(s_1, \cdots, s_n) = \max_{b \in O} v_{k \neq i}(b, t_k),$$

also das zweithöchste Gebot.

□

Lemma 2.3.21. *Die Vickrey-Auktion ist anreizkompatibel und individuell rational.*

Beweis. Die Vickrey Auktion ist ein VCG-Mechanismus mit Clarke-Pivot-Regel. □

Lemma 2.3.22. *Die Vickrey-Auktion besitzt ein dominantes Gleichgewicht (siehe Definition 2.3.13).*

Beweis. Beweis siehe [NRTV07]. □

Betrachtet man die Vickrey-Auktion in Hinblick auf die in Abschnitt 2.3.7 definierten Komplexitätseigenschaften, gibt es im Wesentlichen zwei Schwachstellen [Ste08]:

- Typbestimmung und
- Ergebnisbestimmung.

Die Typbestimmung kann je nach Informationsstruktur des Bieters unterschiedlich komplex sein. Der tatsächliche Wert eines Objekts ist nicht immer offensichtlich, beispielsweise ist der Wert einer Funklizenz abhängig vom erwarteten Ertrag [Kwi05].

Die Ergebnisbestimmung kann ein kombinatorisches Optimierungsproblem sein, wenn in der Auktion unterschiedliche Objekte angeboten werden. Ein Bieter kann eventuell Gebote für Kombinationen von Objekten abgeben. Dies führt zu einem dem Knapsack-Problem [Pap93] ähnlichen Problem. Prinzipiell lässt sich dies nicht ohne Verlust der Eigenschaften der VCG-Mechanismen approximieren [Ste08].

Die Strategiebestimmung ist in einer Vickrey-Auktion trivial, da hier nur der ermittelte wahrheitsgemäße Typ deklariert wird. Die Kommunikationskomplexität ist sehr gering, da in der Vickrey-Auktion nur der Bieter über die Auktion informiert werden muss und anschließend der Auktionator den Bieter über das Ergebnis benachrichtigen muss [Ste08].

Die Vickrey-Auktion hat allerdings im Allgemeinen auch einige Nachteile [San96, Rot07], die für eine tatsächlich realisierte Auktion beachtet werden sollten:

- Eine budget-ausgeglichene Vickrey-Auktion, in der Verkäufer mit betrachtet werden, ist im allgemeinen Fall entweder individuell rational und nicht effizient oder nicht individuell rational und effizient [MS83]. Ist der Wert eines Objekts für den Verkäufer höher als für den Käufer, gibt es zwei Möglichkeiten. Entweder der Verkäufer verkauft das Objekt trotzdem. Dadurch hat der Verkäufer einen negativen Nutzen, die Auktion ist also nicht individuell rational. Oder der Verkäufer verkauft das Objekt nicht, dann ist das Ergebnis nicht effizient, da das Objekt nicht verkauft wurde.

- Ein Agent kann seinen wahren Wert nicht immer bieten, da er nicht immer über ein geeignet großes Budget verfügt.

- Das dominante Gleichgewicht ist in der Vickrey-Auktion ein schwaches Gleichgewicht. Ist einem der Agenten bekannt, dass er mit einem bestimmten Wert nicht den Zuschlag bekommt, kann dieser Agent alternativ auch kein Gebot abgeben, da es seinen Nutzen nicht verändert. Wenn der Agent ansonsten das zweit höchste Gebot abgegeben hätte, verringert dies das Einkommen des Verkäufers.

- Die Vickrey-Auktion sorgt dafür, dass Agenten den wahren Wert deklarieren und damit private Informationen preisgeben. Die Preisgabe kann für einen Agenten unerwünscht und nachteilig sein, da dadurch andere Agenten private Informationen des Agenten erfahren.

- Die Vickrey-Auktion ist anfällig gegenüber Manipulationen:
  - Der Verkäufer kann lügen und einen höheren Preis als zweit höchstes Gebot ausgeben.
  - Die Käufer können durch Absprachen den Preis verringern.

- In wiederholten Auktionen kann ein Agent abwägen, wann er an einer Auktion teilnimmt. Beispielsweise kann der zu zahlende Preis für ein gleichwertiges Objekt in einer weiteren Auktion auf Grund geringerer Konkurrenz niedriger ausfallen. Ist der zu erwartende Preisunterschied größer als die Kosten der Wartezeit, ist es für den Agent besser in der ersten Auktion nicht seinen wahren Wert zu bieten. Der schlimmste Fall tritt auf, wenn alle Agenten vollständig geduldig sind. In diesem Fall spielt es für den Agent keine Rolle, wann er einen Zuschlag erhält. Wird das Spiel unendlich oft durchgeführt, werden alle Agenten 0 bezahlen, oder falls vorhanden den Reservierungspreis.

2.4 Gruppenkommunikation

Ein Gruppenkommunikations-System unterstützt die Kommunikation zwischen verteilten Prozessen. Die Kommunikation erfolgt als 1-zu-n Kommunikation, d.h. jeder Prozess kann Nachrichten an alle anderen Prozesse senden.

Eine Gruppe ist eine Menge von Prozessen, die Mitglieder dieser Gruppe sind. Eine Gruppe besitzt immer einen logischen Namen, mit Hilfe dessen die einzelnen Prozesse Nachrichten an die Gruppe versenden. Der Gruppenkommunikationsmechanismus sendet diese Nachricht anschließend an alle Gruppenmitglieder.

Eine Art von Gruppenkommunikations-Mechanismen sind Sichten-orientierte Systeme, in denen ein Mitglieder-Service eine Liste mit aktiven Mitgliedern der Gruppe verwaltet. Die Liste, welche die Ausgabe des Mitglieder-Service ist, wird als Sicht bezeichnet.

Die hier im weiteren Verlauf dargestellten Eigenschaften von Gruppenkommunikations-Systeme entstammen der Studie [CKV01].

2.4.1 Übersicht

Um sinnvoll nutzbar zu sein, muss ein Gruppenkommunikations-System bestimmte Eigenschaften erfüllen. Dazu wird ein unendlicher Trace einer Ausführung des Systems betrachtet und mit Hilfe dieses Traces werden Sicherheits- und Lebendigkeitseigenschaften definiert.

Zuerst werden generelle Interaktionen zwischen Anwendung und Gruppenkommunikations-System beschrieben. Darauf aufbauend wird auf Sicherheitseigenschaften des Mitglieder-Service und der Nachrichtenübertragung eingegangen. Anschließend wird diskutiert, wann Nachrichten aus dem System entfernt werden können und in welcher Reihenfolge Nachrichten bei den einzelnen Prozessen ankommen sollen.

Im Anschluss daran werden erwünschte Lebendigkeitseigenschaften des Systems dargestellt.

Ein Gruppenkommunikations-System interagiert mit der Anwendung mittels:

- Senden von Nachrichten
- Empfangen von Nachrichten
- Änderung der Sicht

- Benachrichtigung, dass eine Nachricht stabil ist. Eine Nachricht ist immer dann stabil, wenn sie an alle Gruppenmitglieder ausgeliefert wurde. Die Nachricht kann also gelöscht werden.

Außerdem interagiert das System selbst mit der Umgebung:

- Ein Prozess kann abstürzen.
- Ein Prozess kann wiederhergestellt werden.

Ein Trace ist eine Abfolge von Interaktionen mit der Anwendung oder der Umgebung.

2.4.2 Mitglieder-Service

Der Mitglieder-Service muss einige grundsätzliche Eigenschaften erfüllen:

- Eine Sicht wird durch einen Prozess nur dann installiert, wenn der Prozess selbst darin enthalten ist.
- Es werden nur neuere Sichten installiert.
- Bevor eine Nachricht empfangen werden kann, muss eine Sicht installiert werden.
- Wird die Gruppe partitioniert, gibt es zwei Möglichkeiten. Entweder es wird nur eine einzige primäre Partition weitergeführt oder die Gruppe ist partitionierbar und jede Partition wird für die Zeitdauer der Partitionierung eigenständig. Sobald die Partitionierung nicht mehr vorliegt, müssen auf jeden Fall die Zustände der einzelnen Prozesse wieder abgeglichen werden.

2.4.3 Nachrichtenauslieferung

Konsistenz in der Gruppenkommunikation betrifft zwei unterschiedliche Eigenschaften:

- Nachrichtenauslieferung
- Nachrichtenreihenfolge

Die Nachrichtenauslieferung, also der Versand und Empfang von Nachrichten, betrachtet, welche Gruppenmitglieder welche Nachrichten versenden bzw. empfangen. Die Nachrichtenreihenfolge betrachtet hingegen in welcher Reihenfolge Nachrichten bei einem Gruppenmitglied ausgeliefert werden.

Eine wichtige Konsistenzeigenschaft ist folglich, dass das Gruppenkommunikations-System die Nachrichten so ausliefert, dass möglichst alle Prozesse die gleichen Nachrichten erhalten. Dabei gelten immer mindestens die zwei folgenden Voraussetzungen:

- Eine Nachricht muss, bevor sie empfangen werden kann, gesendet werden.
- Eine Nachricht darf nur einmal von jedem Empfänger empfangen werden.

Es gibt zwei grundsätzliche Eigenschaften, wann eine Nachricht tatsächlich ausgeliefert werden soll:

- Die Nachricht muss in der selben Sicht empfangen werden, in der sie gesendet wurde
- oder die Nachricht muss in jedem Prozess, der die Nachricht empfängt, in der selben Sicht empfangen werden.

Bietet das Gruppenkommunikations-System die erste Eigenschaft, wird garantiert, dass der sendende und die empfangenden Prozesse alle die gleiche Sicht auf die Gruppe haben. Dies sichert die Konsistenz der Nachrichtenübertragung. Die zweite Eigenschaft ist schwächer und garantiert nur Konsistenz über alle empfangenden Prozesse hinweg.

Eine mächtige Erweiterung dieser Konzepte ist die virtuelle Synchronität:

- In einem Gruppenkommunikationsmechanismus mit virtueller Synchronität werden Sichten auf allen Prozessen gleichzeitig installiert, d.h. wenn ein Prozess eine neue Sicht installiert hat, ist garantiert, dass auch alle anderen Prozesse diese Sicht besitzen. Dies hat zur Folge, dass Nachrichten immer von allen Prozessen in der gleichen Sicht empfangen werden.

Da Nachrichten verloren gehen können, müssen Nachrichten auch nach der ersten Übertragung vorgehalten werden, so dass diese erneut übertragen werden können. Es muss deshalb geregelt werden, wann eine Nachricht als für alle Prozesse bekannt gilt und folglich gelöscht werden kann. Dies lässt sich durch Stabilitäts-Mitteilungen erreichen, die versendet werden, sobald die Nachrichten bei einem Prozess empfangen wurde. Sobald alle Prozesse den Empfang bestätigt haben, ist die Nachricht stabil, d.h. die Nachricht kann bei allen Prozessen gelöscht werden.

2.4.4 Nachrichtenreihenfolge

Die Nachrichtenreihenfolge spielt ebenfalls eine wichtige Rolle, sie kann je nach Anforderung der Anwendung unterschiedlich starken Anforderungen genügen:

- Nutzt das System eine FIFO-Ordnung, werden die Nachrichten von einem Prozess bei allen anderen in der gleichen Reihenfolge empfangen wie, sie ursprünglich versendet wurden.
- Bei der kausalen Ordnung werden kausale Zusammenhänge zusätzlich mit beachtet. Prozess 1 sendet eine Nachricht, Prozess 2 empfängt diese Nachricht und sendet eine Antwort darauf. Ein dritter Prozess sollte zuerst die Nachricht von Prozess 1 empfangen und erst danach die Nachricht von Prozess 2, da die beiden Nachricht in einem kausalen Zusammenhang stehen.
- Die totale Ordnung ist eine noch restriktivere Eigenschaft. Jeder Prozess empfängt alle Nachrichten, die von allen Prozessen gesendet werden, in der gleichen Reihenfolge.

2.4.5 Lebendigkeit

Ohne die Eigenschaft der Lebendigkeit können alle bisherigen Eigenschaften trivial implementiert werden, indem das Gruppenkommunikations-System keine Arbeit verrichtet. Darum müssen geeignete Eigenschaften definiert werden, die einerseits erfüllbar sind, auf der anderen Seite allerdings nicht trivial implementierbar sind.

Eine wichtige Komponente in diesem Zusammenhang ist ein Fehlerdetektor. Da in einem Netzwerk in dem Nachrichten verloren gehen können und auch keine Obergrenze für die Dauer der Nachrichtenübertragung bekannt ist, kein perfekter Fehlerdetektor möglich ist [CHTCB96, CT96, CKV01], müssen geeignete schwächere Eigenschaften für den Fehlerdetektor gefunden werden:

- Ein stabiles Teilsystem besteht dann, wenn es einen Zeitraum gibt, in welchem eine Menge von Prozessen lebten und alle miteinander verbunden sind. Andere Prozesse dürfen keine Verbindung zu diesen Prozessen besitzen.
- Existiert ein solches Teilsystem, ist ein eventuell perfekter Fehlerdetektor möglich, d.h. irgendwann liefert der Fehlerdetektor die Prozesse des stabilem Teilsystems als die Menge der lebendigen Prozesse zurück.

Sind diese zwei Voraussetzungen erfüllt, d.h. es existiert ein stabiles Teilsystem und ein solcher Fehlerdetektor, kann ein Gruppenkommunikations-System folgende Eigenschaften implementieren:

- Jeder Prozess installiert irgendwann die Sicht des stabilen Teilsystems.
- Jeder Prozess des stabilen Teilsystems empfängt alle gesendeten Nachrichten, die von einem Prozess aus der stabilen Komponente gesendet wurden.
- Jeder Prozess stellt den Empfang aller selbst versendeten Nachrichten sicher, außer der versendende Prozess ist abgestürzt.
- Jede Nachricht, die von einem Prozess versendet wurde, wird irgendwann stabil und kann gelöscht werden.

2.4.6 Beispiel - JGroups

JGroups ist ein Gruppenkommunikationsmechanismus, dessen Eigenschaften durch eine flexible Konfiguration verändert werden können [Ban98]. Dadurch lässt sich JGroups leicht an verschiedene Anforderungen und Umgebungen anpassen. Architektur und Eigenschaften werden basierend auf dem ursprünglichen Entwurf in [Ban98] und der aktuellen Dokumentation [Ban08a] dargestellt.

Architektur

Die Architektur von JGroups lässt sich in verschiedene Schichten einteilen (Abbildung 2.2). Die Anwendung kommuniziert entweder mit einem sogenannten Building-Block oder direkt mit einem Channel.

Ein Building-Block ist eine Komponente, die abstraktere Funktionen anbietet, beispielsweise eine verteilte Hashtable (DHT) oder einen Wahlalgorithmus. Die Komponenten führen ihre Funktion gegenüber der Anwendung transparent durch, d.h. z.B. eine DHT wird, nachdem etwas von der Anwendung daran geändert wurde, automatisch an alle Gruppenmitglieder repliziert oder automatisch eine Wahl durchgeführt.

Alternativ kann die Anwendung auch direkt auf einen Channel zugreifen, der Funktionalität ähnlich zu TCP-Sockets bereitstellt. Der Channel selbst nutzt einen Protokollstack, welcher flexibel konfiguriert werden kann, so dass das Gruppenkommunikations-System unterschiedliche Eigenschaften aufweist, beispielsweise virtuelle Synchronität oder eine

bestimmte Nachrichtenreihenfolge. Auch das Protokoll zur Netzwerkübertragung kann an die Umgebung angepasst werden.

Eigenschaften

JGroups ist ein Sichten-orientiertes Gruppenkommunikations-System, das je nach Konfiguration des Protokollstacks unterschiedliche Eigenschaften aufweist.

Das GMS-Protokoll implementiert den Mitglieder-Service, welcher durch weitere Protokolle ergänzt werden kann:

- Es werden nur Sichten installiert, in denen der Prozess selbst enthalten ist.
- Es werden nur neuere Sichten installiert.
- Mit dem VIEW_ENFORCER-Protokoll wird weiterhin garantiert, dass einer Anwendung durch JGroups keine Nachrichten ausgeliefert werden, solange keine Sicht installiert ist.
- Bei Partitionierungen einer Gruppe kann entweder die Primäre Partition weiter betrieben werden, oder es ist die Weiterführung aller Partitionen der Gruppe möglich, die bei einem Zusammenschluss ihren Zustand mit Hilfe eines MERGE-Protokolls synchronisieren müssen.

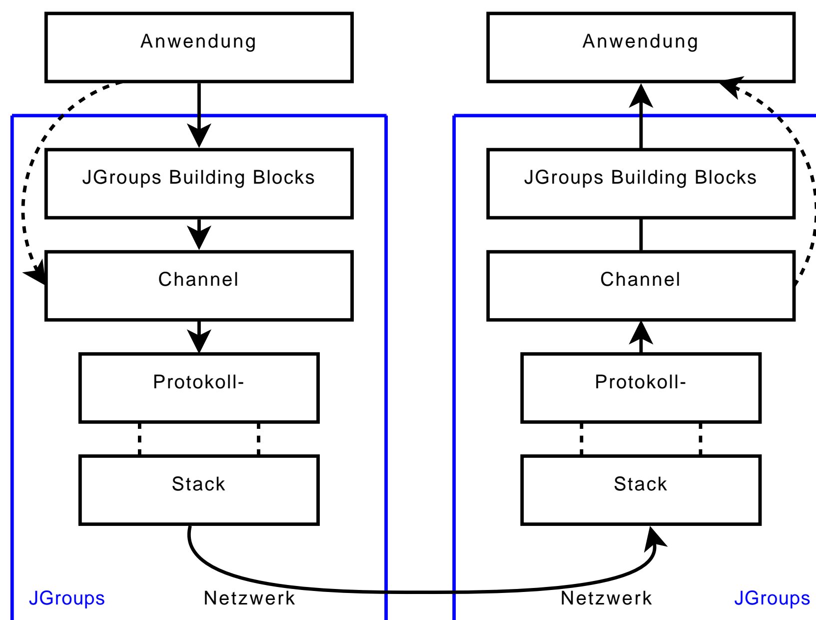

Abbildung 2.2: Die Architektur von JGroups.

Standardmäßig bietet JGroups keine Garantien bezüglich der Nachrichtenauslieferung in Sichten, d.h. es wird nicht garantiert, dass Nachrichten nur in der gleichen Sicht empfangen werden und auch nicht, dass Nachrichten nur in der gleichen Sicht gesendet und empfangen werden können. Der Protokollstack kann allerdings mit dem `FLUSH`-Protokoll so verändert werden, dass JGroups die Eigenschaft der virtuellen Synchronität aufweist.

Als weitere Eigenschaft lässt sich die Nachrichtenreihenfolge verändern:

- Das `NAKACK`-Protokoll resultiert in einer FIFO-Ordnung.
- Das `CAUSAL`-Protokoll resultiert in einer kausalen Ordnung.
- Das `SEQUENCER`-Protokoll resultiert in einer totalen Ordnung

Die Lebendigkeitseigenschaften werden im wesentlichen durch den Fehlerdetektor festgelegt. Alle Fehlerdetektoren sind im oben definierten Sinne eventuell perfekte Fehlerdetektoren, mit der Einschränkung, dass die Fehlerdetektoren auch Falschmeldungen erzeugen können, wenn Prozesse zu langsam agieren, da die Fehlerdetektoren nach einer einstellbaren Zeitspanne automatisch einen ausgefallenen Prozess melden. JGroups kann alle vier Lebendigkeits-Eigenschaften erfüllen, indem ein beliebiger Fehlerdetektor, das `NAKACK`-Protokoll und das `STABLE`-Protokoll kombiniert werden.

Alle bisher beschriebenen Eigenschaften sind unabhängig vom unterlagerten Transport.

Protokolldetails

Die Umgebung, in der JGroups eingesetzt werden soll, beeinflusst weitere Eigenschaften von JGroups. Die wesentlichsten Punkte betreffen

- die Auswahl des Fehlerdetektors und
- mögliche Transportprotokolle.

Für die Auswahl des Fehlerdetektors spielt die Anwendung eine entscheidende Rolle. Falls ein anwendungsspezifischer Fehlerdetektor genutzt werden kann, ist es möglich, dass der Fehlerdetektor bessere Eigenschaften aufweist. Standardmäßig bietet JGroups folgende Fehlerdetektoren an:

- Der `FD`-Fehlerdetektor sendet über JGroups jeweils zum nächsten Nachbar eine Anfrage und wartet auf die Antwort. Der Fehlerdetektor erfolgt also ringförmig.

JGroups verwaltet dazu für jede Sicht eine Liste der bekannten Gruppenmitglieder. Der nächste Nachbar ist der Nachfolger in dieser Liste. Das letzte Mitglied in dieser Liste hat das erste Mitglied als Nachbar.

- Der `FD_SOCK`-Fehlerdetektor ist ähnlich zum `FD`-Fehlerdetektor, baut allerdings separate Verbindungen über zusätzliche Sockets auf.
- Mit dem `FD_ALL`-Fehlerdetektor sendet jeder Prozess in einer Gruppe periodisch eine Nachricht an alle Gruppenmitglieder.

Der Aufbau eines Rings hat den Vorteil, dass die Menge der zu sendenden Nachrichten deutlich reduziert wird. Allerdings konvergieren die Sichten der einzelnen Prozesse dadurch langsamer zur stabilen Komponente, falls mehrere benachbarte Prozesse nahezu zeitgleich ausfallen. Der `FD_ALL`-Fehlerdetektor hat dagegen den offensichtlichen Nachteil, dass das Nachrichtenaufkommen quadratisch anwächst und damit ab einer bestimmten Gruppengröße den Netzwerktransport überlastet.

Die möglichen Transportprotokolle sind durch die Eigenschaften des unterlagerten Netzwerks vorgegeben. Prinzipiell unterstützt JGroups:

- IP-Multicast,
- UDP und
- TCP.

Ein Spezialfall ist das `SHARED_LOOPBACK` Transportprotokoll, welches beispielsweise für Simulationszwecke eingesetzt werden kann. Es handelt sich dabei um ein virtuelles Transportprotokoll, das innerhalb eines Prozesses Nachrichten ohne ein unterlagertes, reales Netzwerk zwischen einzelnen Gruppenmitgliedern verteilt.

Programmierschnittstelle

Die Programmierschnittstelle von JGroups kann in vier Teile aufgeteilt werden:

- Erzeugung und Konfiguration eines Channels
- Kommunikation über einen Channel
- Protokoll-Schnittstelle
- Building Blocks

Konfiguration und Kommunikation Die Erzeugung und Konfiguration eines Channels, sowie die Kommunikation über einen Channel wird prinzipiell mit Hilfe der Klasse `JChannel` durchgeführt. Dem `JChannel`-Konstruktor wird bei der Erzeugung eines neuen JChannels die Konfiguration des Protokollstacks übergeben. Die Konfiguration kann entweder über eine Konfigurationsdatei erfolgen, oder über einen String. Beide Methoden unterscheiden sich im Wesentlichen durch die Syntax.

Ein möglicher Konfigurations-String ist beispielsweise:

```
TCP(start_port=10000;
    loopback=true;
    recv_buf_size=20000000;
    send_buf_size=640000;
    sock_conn_timeout=300):
TCPPING(timeout=5000;
        initial_hosts=vs23[10000],vs24[10000];
        port_range=1;
        num_initial_members=3):
MERGE2(max_interval=10000;
       min_interval=5000):
FD(timeout=10000;
   max_tries=3):
VERIFY_SUSPECT(timeout=15000):
pbcast.NAKACK(max_xmit_size=60000;
              use_mcast_xmit=false;
              gc_lag=0;
              discard_delivered_msgs=true;
              retransmit_timeout=100,200,300,600,1200,2400,4800):
pbcast.STABLE(stability_delay=1000;
              desired_avg_gossip=50000;
              max_bytes=400000):
pbcast.GMS(print_local_addr=true;
           join_timeout=3000;
           join_retry_timeout=2000;
           shun=true)
pbcast.FLUSH
```

Listing 2.1: Beispielkonfiguration für einen JGroups-Protokoll-Stack

Als erstes fällt auf, dass der JGroups-Protokoll-Stack umgekehrt zu gewöhnlichen Protokoll-Stacks konfiguriert wird, d.h. das Transportprotokoll ist das erste Protokoll, das eigentlich am nächsten an der Benutzerschicht liegende Protokoll das letzte.

Das erste Protokoll ist folglich `TCP`, es wird das TCP-Protokoll auf Netzwerkebene verwendet. Hier wird der genutzte Port, Puffer zum Versenden und Empfangen und ein Timeout für den genutzten TCP-Socket festgelegt. Eine Besonderheit ist hier die Option `loopback`. Durch sie wird geregelt, ob ein Mitglied sich Nachrichten auch selbst zustellt.

TCPPING ist dafür zuständig die initialen Mitglieder der Gruppe zu finden. Bei TCP sollten hier alle bekannten Mitglieder definiert werden. TCP bietet im Gegensatz zu IP Multicast keine Möglichkeit Gruppenmitglieder zu finden, die nicht mindestens einem Mitglied bekannt sind.

Das MERGE2 Protokoll sorgt dafür, dass nach einer Partitionierung wieder alle Partitionen zur gleichen Sicht auf die Gruppe finden.

FD und VERIFY_SUSPECT sind Teil des Fehlerdetektors. FD ist der schon beschriebene Fehlerdetektor. Ein Fehlerdetektor generiert sogenannte SUSPECT-Nachrichten falls er den Defekt eines Mitglieds festgestellt hat und sendet diese an alle Gruppenmitglieder. Jedes Gruppenmitglied nutzt anschließend das VERIFY_SUSPECT-Protokoll, um festzustellen ob das verdächtigte Mitglied tatsächlich ausgefallen ist oder nur temporär nicht erreichbar war.

Das NAKACK-Protokoll ist für den zuverlässigen Nachrichtenversand zuständig. Allen Nachrichten wird eine monoton aufsteigende Sequenznummer zugewiesen vom Sender. Sind Nachrichten auf dem Weg zu einem Empfänger verloren gegangen, fordert der Empfänger die fehlende Nachricht beim Sender an, welcher die Nachricht an den Empfänger wiederholt versendet. Die Nutzung des NAKACK-Protokoll resultiert außerdem in einer FIFO-Ordnung der Nachrichten beim Empfänger.

Das STABLE-Protokoll übernimmt den Versand von Stabilitätsnachrichten und das löschen von als stabil bekannten Nachrichten.

Das GMS-Protokoll ist der Mitglieder-Service. Dieses Protokoll verwaltet die Sicht eines Mitglieds auf die Gruppe. Hier werden neue Mitglieder hinzugefügt und Mitglieder welche die Gruppe verlassen haben oder ausgefallene sind gelöscht.

Das FLUSH-Protokoll sorgt dafür, dass die Auslieferung von Nachrichten an die Anwendung, sowie der Versand von Nachrichten durch die Anwendung während Änderungen der Sicht blockiert ist. Dies resultiert in der oben genannten virtuellen Synchronität.

Nachdem ein JChannel-Objekt erzeugt wurde, muss dieses im Anschluss durch einen Aufruf von JChannel.connect() mit der eigentlichen Gruppe verbunden werden. Dem Aufruf wird der Gruppenname der Gruppe übergeben, zu der eine Verbindung hergestellt werden soll.

Anschließend kann das JChannel-Objekt zum Senden und Empfangen von Nachrichten genutzt werden. Das Senden kann direkt über das JChannel-Objekt erfolgen. Für den Empfang muss eine von der Receiver-Schnittstelle abgeleitete Klasse genutzt werden,

welche alle Nachrichten, die mittels des `JChannel`-Objekts empfangen werden, zugestellt bekommt.

Protokollschnittstelle JGroups kann um weitere Protokolle erweitert werden. Dazu muss die `Protocol`-Klasse erweitert werden. Die Wesentlichen zu implementierenden Methoden sind `Protocol.down()` und `Protocol.up()`. Die `down()`-Methode wird vom im Protokoll-Stack darüberliegenden Protokoll aufgerufen und sollte falls nötig die `down()`-Methode des nächsten Protokolls aufrufen. Die `up()`-Methode verhält sich entgegengesetzt, wird also vom darunterliegenden Protokoll aufgerufen und sollte das darüberliegende Protokoll benachrichtigen.

Ein Transportprotokoll muss statt der `Protocol`-Klasse die `TP`-Klasse erweitern. Die `TP`-Klasse erweitert die `Protocol`-Klasse unter anderem um zwei zusätzlichen Methoden, `TP.sendToAllMembers()` und `TP.sendToSingleMember()` die zum Nachrichtenversand genutzt werden.

Building-Blocks Jeder Building-Block besitzt seine eigene spezifische Schnittstelle zur Anwendung. Da in der vorliegenden Arbeit nur der `RpcDispatcher`-Building-Block genutzt wird, wird dieser hier beispielhaft dargestellt.

Ein `RpcDispatcher`-Objekt wird durch den Aufruf eines Konstruktors dieser Klasse erzeugt. Dabei werden dem Konstruktor ein zuvor erzeugtes `JChannel`-Objekt und ein beliebiges Objekt übergeben. Das `JChannel`-Objekt wird für die Kommunikation genutzt, das zweite Objekt wird für RPC-Aufrufe von anderen Mitgliedern der Gruppe genutzt.

Zum Durchführen von RPC-Aufrufen besitzt die `RpcDispatcher`-Klasse zwei Möglichkeiten an:

- `callRemoteMethod()` und
- `callRemoteMethods()`.

Die erste Methode führt einen RPC-Aufruf bei genau einem entfernten Mitglied durch. Die zweite Methode nutzt einen Multicast, um bei allen Mitgliedern der Gruppe die gleiche Methode aufzurufen, also einen Multicast-RPC durchzuführen. Falls gewünscht, können beide Methoden den Rückgabewert zurückliefern. Der Multicast-RPC kann so konfiguriert werden, dass entweder ein einziger Rückgabewert, alle Rückgabewerte oder nur

alle bis zu einem bestimmten Timeout eingetroffene Rückgabewerte zurückgeliefert werden. Der einfache RPC liefert den Rückgabewert als `Object` zurück, der Multicast-RPC in einem `RspList`-Objekt, welches zusätzlich Informationen enthält, ob Mitglieder ausgefallen sind.

Kapitel 3

Analyse

3.1 Problemstellung

Die Arbeit betrachtet ein Softwaresystem, bestehend aus einer Menge von Services und einer Menge von Workflows, welche die Services zur Abarbeitung ihrer einzelnen Aktivitäten nutzen. Für jeden Workflow werden durch ein SLA bestimmte Dienstgüte-Ziele vereinbart, die ein Workflow einhalten muss. Wenn die Dienstgüte-Ziele nicht eingehalten werden, muss ein Management-System eingreifen und geeignete Aktionen durchführen, um die geforderte Dienstgüte wiederherzustellen. Ähnliche Systeme sind beispielsweise in [YBT05, DDK$^+$04, ZQS05] beschrieben.

[YBT05] definiert ein System als gerichteten azyklischen Graph. Der Graph beschreibt jeweils den Kontrollfluss eines Workflows. Die Zuordnung von Aktivitäten eines Workflows zu den davon genutzten Services bleibt als Scheduling-Problem offen. Dazu muss eine Menge von gleichartigen Services vorhanden sein, von denen jeweils ein Service ausgewählt wird, der das geforderte SLA einhalten kann. Bei Nichteinhaltung des SLAs wird ein neuer Schedule berechnet.

[DDK$^+$04] beschreibt ein Managementsystem für Webservices [ACKM04], das die Ressourcenzuteilung dynamisch ändern kann, um SLAs einhalten zu können. Das System enthält verschiedene Aktivitäten. Zuerst werden zwischen Nutzer und Service SLAs ausgehandelt. Anschließend werden dem Nutzer Ressourcen zugeteilt, so dass das SLA erfüllt werden kann. Die tatsächliche Nutzung der Ressourcen wird während der Ausführung der Webservices überwacht und auf Grund von Messwerten wird die Zuteilung dynamisch verändert.

[ZQS05] definiert keine Workflows. Stattdessen wird ausschließlich die Service-Ebene

mit den damit verbundenen Ressourcen definiert. Zwischen dem nicht näher beschriebenen Nutzer und den Services werden SLAs für jeden angeforderten Service ausgehandelt. Wenn ein SLA verletzt wird, wird versucht, durch Kapazitätsänderungen die SLA-Verletzung zu beheben. Im Wesentlichen wird ein Ersatz-Service gesucht, der die Anforderungen des SLAs erfüllen kann.

Die meisten bisherigen Entwicklungen verwenden zentrale Algorithmen, um die Einhaltung der SLAs zu überwachen und gegebenfalls korrigierend einzugreifen. Diese Verfahren stoßen bei sehr großen Systemen auf Grund der stark zunehmenden Komplexität an Grenzen.

Zentrale Algorithmen können ein weiteres Problem gar nicht lösen. Jeder Service kann sich in einer anderen administrativen Domänen befinden. Jede administrative Domäne möchte natürlich den eigenen Nutzen maximieren und den Einfluss von anderen auf die eigenen Services minimieren. Eine mögliche Lösung dieses Problems ist die Verwendung dezentraler Algorithmen auf Basis des in Abschnitt 2.3 beschriebenen Mechanismus-Designs.

Sogenannte rationale Agenten des Mechanismus-Design modellieren eigennütziges Verhalt von Agenten (vgl. Abschnitt 2.3.3), dies entspricht dem Verhalten einer administrativen Domäne, die wie ein rationaler Agent den eigenen Nutzen erhöhen will. Mögliche Mechanismen sind beispielsweise Auktionen (vgl. Abschnitt 2.3.9). Eine mögliche Auktionsform mit interessanten theoretischen Eigenschaften, eine generalisierte Form der Vickrey-Auktion, wird hier in der Arbeit untersucht.

3.2 Systemdefinition

Im Folgenden werden ein service-orientiertes System s und seine Eigenschaften formal definiert:

Definition 3.2.1. Ein Service-orientiertes System besteht aus einem Tupel

$s = (W, A, is_contained, E, S, allocates)$:

$W = \{w_1, \cdots, w_i\}$	einer endlichen Menge von *Workflows*
$A = \{a_1, \cdots, a_j\}$	einer endlichen Menge von *Aktivitäten*
$is_contained : A \rightarrow W$	jede Aktivität ist in einem Workflow enthalten
$E \subseteq A \times A$	einer endlichen Menge von Verbindungen zwischen Aktivitäten
$S = \{s_1, \cdots, s_k\}$	einer endlichen Menge von *Services*
$allocates : A \rightarrow S$	einer Abbildung von Aktivitäten auf Services

Weiterhin ist die Menge E einzuschränken:

$$\forall e(e = (a, a') \Rightarrow is_contained(a) = is_contained(a'))$$

Aktivitäten sind nur innerhalb eines Workflows verbunden und können somit keine Abhängigkeiten zwischen Workflows modellieren.

Definition 3.2.2. Die Mengen A und E bilden zusammen einen gerichteten azyklischen Graph $G = (A, E)$, wobei A die Menge der Knoten und E die Menge der Kanten ist.

Eine Kante $(p, c) \in E$ besteht aus einer *Eltern-Aktivität* p und einer *Kind-Aktivität* c. Weiterhin besitzt der Graph folgende induktiv definierte Struktur:

(IA) Ein Aktivität a ist eine *atomare Aktivität* $a \in A$.

(IS) Es gibt weiterhin *Zusammengesetzte Aktivitäten*:

- Sind $a_1, \cdots, a_i$ Aktivitäten, so ist auch eine *Sequenz* $a_1, \cdots, a_i \in \mathfrak{s}$ eine Aktivität. Diese besteht aus einer Verkettung von i Aktivitäten:
  - Die erste Aktivität der Sequenz a_1 besitzt keine Eltern-Aktivität $p_1 \in A$ und nur eine Kind-Aktivität $c_1 = a_2$.
  - Jede weitere Aktivität, $a_2, \cdots, a_{i-1}$ hat nur eine Eltern-Aktivität $p_j = a_{j-1}$ und nur eine Kindaktivität $c_j = a_{j+1}$.
  - Die letzte Aktivität a_i besitzt nur eine Eltern-Aktivität $p_i = a_{i-1}$ und keine Kind-Aktivität $c \in A$.
- Sind $a_1, \cdots, a_i$ Aktivitäten, dann ist auch eine *Parallelausführung* $a_1, \cdots, a_i \in \mathfrak{p}$ eine Aktivität. Jede Aktivität $a_1, \cdots, a_i$ besitzt die gleiche Eltern- und Kind-Aktivität, d.h. $p_1 = \cdots = p_i \wedge c_1 = \cdots = c_i$.

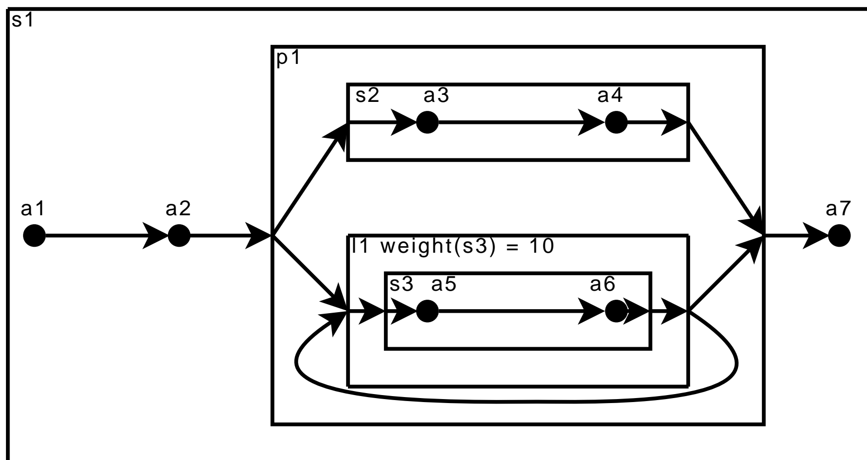

Abbildung 3.1: Ein Beispielworkflow mit mehreren Sequenzen, einer Parallelausführung und einer Schleife.

- Ist a eine Aktivität, so ist auch eine *Schleife* $a \in \mathfrak{l}$ eine Aktivität. Die Schleife weist der Aktivität a mittels einer Gewichtsfunktion

  $$weight_{\mathfrak{l}}(a) =_{def} n$$

  ein Gewicht $n \in \mathbb{N}$ zu, welches die maximale Anzahl von Schleifendurchläufen beschreibt. Die azyklische Eigenschaft des Graph wird hiervon nicht verändert, da die Schleife in jedem Fall ausgerollt werden kann, da eine Maximalanzahl von Schleifendurchläufen bekannt sein muss.

Nach den Definitionen 3.2.2 lassen sich Sequenzen, Parallelausführungen und Schleifen beliebig schachteln. Beispielsweise ist Abbildung 3.1 ein korrekter Workflowgraph mit insgesamt drei Sequenzen ($s1 = \{a3, a4\}$, $s2 = \{a5, a6\}$, $s3 = \{a1, a2, p1, a7\}$), einer Parallelausführung ($p1 = \{s1, l1\}$) und einer Schleife ($l1 = s2$). Die Darstellung des

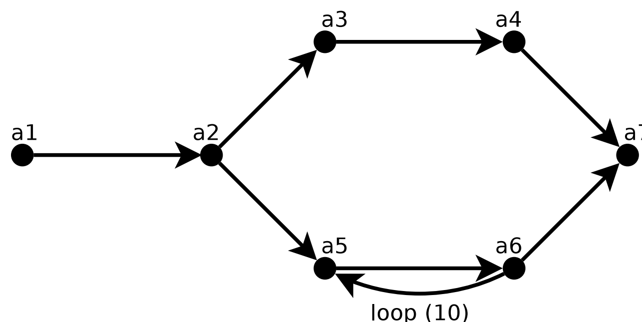

Abbildung 3.2: Der identische Workflow wie in Abbildung 3.1 in einer anschaulicheren Darstellung.

Workflows aus Abbildung 3.1 lässt anschaulicher wie in Abbildung 3.2 darstellen.

3.3 Dienstgüte-Eigenschaften

Das in Abschnitt 3.2 definierte Service-orientierte System wird wie folgt durch Dienstgüte-Eigenschaften erweitert. Ein SLA (vgl. Abschnitt 2.1) wird für jeden Workflow definiert. Häufig ist ein SLA mehrdimensional, da verschiedene SLOs für verschiedene Dienstgüteparameter definiert wird. Die Einhaltung eines SLAs ist folglich ein mehrdimensionales Problem. Zur Vereinfachung des Problems wird im weiteren Verlauf der Arbeit nur die Einhaltung der Antwortzeit als einziges Teilproblem behandelt, wodurch ein einfacheres, eindimensionales Problem entsteht:

Im Weiteren werden alle Zeitwerte als diskrete, endliche Werte betrachtet. Dies erfolgt aus einerseits aus praktischen Gründen, da einem Rechner nur Zeitmessungen mit endlicher Genauigkeit möglich sind. Im späteren Verlauf der Arbeit wird diese Eigenschaft außerdem ausgenutzt, um jeder diskreten Zeiteinheit einen Wert zuzuweisen. Die Nutzung von diskreten Werten führt teilweise zu numerischen Problemen, durch welche die Summe aller SLO-Zielwerte nicht exakt dem SLA-Zielwert entsprechen muss, sondern tatsächlich kleiner ist. Außerdem muss bei der Aufteilung des SLA-Zielwerts gerundet werden, um wieder einen diskreten Wert zu erhalten.

Definition 3.3.1. SLA- und SLO-Zielwerte werden als diskrete, endliche Werte aufgefasst. Ein SLA-Zielwert beschreibt dabei die maximale Gesamtantwortzeit eines Workflows, ein SLO die maximale Antwortzeit einer Aktivität. Es gibt folglich eine bijektive Abbildung

$$f : SLA \to \mathbb{N},$$

wobei $SLA =$ Menge aller möglichen SLA-Zielwerte. Für die Menge aller SLO-Zielwerte SLO gibt es ebenfalls eine bijektive Abbildung:

$$g : SLO \to \mathbb{N}.$$

Definition 3.3.2. Jedem Workflow wird ein SLA-Zielwert zugeordnet, welches die für den gesamten Workflow geforderten Dienstgüte-Eigenschaften beschreibt:

$$sla : W \to SLA.$$

Jeder Aktivität wird ein SLO zugeordnet:

$$slo : A \rightarrow SLO.$$

Der zugeordnete SLA-Zielwert wird auf die einzelnen Aktivitäten des Workflows in Form von SLOs aufgeteilt, so dass folgende Eigenschaften zutreffen:

- Für jeden möglichen Teil-Pfad einer Parallelausführung ist die Summe der SLOs in etwa gleich.
  $$\sum_{i=1}^{j} slo(a_i) \approx \sum_{k=1}^{l} slo(a_k)$$
  wobei a_i und a_k Elemente von zwei unterschiedlichen Pfaden der Länge j bzw. l einer Parallelausführung sind.

- Für jeden möglichen Pfad durch den Workflow muss die Summe aller SLOs gleich oder kleiner dem Workflow-SLA-Zielwert sein:
  $$\sum_{i=1}^{j} slo(a_i) \leq sla(w)$$
  wobei a_i eine Aktivität innerhalb eines Pfads $P = a_1 \cdots a_j$ durch einen Workflow w ist.

- Enthält ein Pfad eine Schleife, wird das tatsächliche für eine atomare Aktivität angewendete SLO durch das Gewicht der Schleife dividiert.

Definition 3.3.3. Der SLA-Zielwert wird initial nach folgendem Algorithmus aufgeteilt:

1. Es wird der längste Pfad durch einen Workflow w gesucht. Jede Aktivität a des längsten Pfads p erhält ein identisches SLO, d.h. $slo(a) = \lfloor \frac{sla(w)}{\#(p)} \rfloor$, wobei $\overline{\#(p)}$ die Länge des längsten Pfads ist.

2. Alle Aktivitäten von gleich langen Pfaden bekommen ebenfalls dieses SLO zugewiesen.

3. Kürzere Pfade können innerhalb einer Parallelausführung entstehen. Für diese Pfade wird die Summe aller SLOs der Aktivitäten des längsten Pfads in der Parallelausführung gebildet und diese Summe über den kürzeren Pfad gleichmäßig verteilt, analog zum Fall für den gesamten Workflow.

4. Der dritte Schritt wird solange rekursiv fortgesetzt (Parallelausführungen können wiederum Sequenzen enthalten, die Parallelausführungen enthalten), bis alle Aktivitäten ein SLO zugewiesen bekommen.

Um die durch die SLAs und SLOs geforderten Dienstgüte-Eigenschaften zu überprüfen, muss die reale Antwortzeit ermittelt und mit den Anforderungen aus den SLOs verglichen werden. In einem realen System ist die Antwortzeit eine Eigenschaft die nur nach der tatsächlichen Ausführung einer Aktivität ermittelt werden kann. In der hier vorliegenden Arbeit wird davon ausgegangen, dass die Antwortzeit über einen kurzen Zeitraum in etwa konstant bleibt, so dass die ermittelten Antwortzeiten für Änderungen des SLOs herangezogen werden können. Der gemessene Wert wird also als Erwartungswert für weitere Messungen genutzt.

Definition 3.3.4. Der Erwartungswert der Antwortzeit eines Services wird für jede Aktivität ermittelt:

$$t_{response} : A \rightarrow \mathbb{N}$$

Definition 3.3.5 (Ziel des Managementsystems)**.** Das Ziel des Systems ist, die Antwortzeit für alle Aktivität a immer geringer als das zugewiesene SLO zu halten:

$$t_{response}(a) < slo(a)$$

Ist dies nicht der Fall, tritt eine SLO-Verletzung auf. Als Maßnahme müssen geeignete Änderungen am System durchgeführt werden, so dass diese Bedingung zu einem späteren Zeitpunkt wieder erfüllt wird.

Das in Definition 3.3.5 definierte Ziel ist eine hinreichende Bedingung, bei der das SLA eines Workflows immer eingehalten wird. Dies ist allerdings keine notwendige Bedingung für die Einhaltung des SLAs. Beispielsweise können einige Aktivitäten eine deutlich geringer Antwortzeit aufweisen, als durch das jeweilige SLO gefordert wird und somit SLO-Verletzungen einiger Aktivitäten eventuell ausgleichen.

3.4 Lösungsansätze

Eine SLO-Verletzung kann behoben werden, wenn durch die durchgeführten Aktionen eine der beiden Resultate hervorgeht:

- Lockerung des SLOs
- Verkürzung der Antwortzeit

Wird das SLO gelockert, kann erreicht werden, dass das SLO wieder eingehalten wird, sofern die Antwortzeit anschließend unterhalb des höheren SLO-Zielwerts liegt. Eine weiter Möglichkeit ist, am Service selbst etwas zu ändern, so dass die Antwortzeit verkürzt wird. Beispielsweise können neue Ressourcen für einen Service beschafft werden.

Führt keine mögliche Aktion zu einer Erfüllung des Ziels aus Definition 3.3.5, muss die SLO-Verletzung an eine höhere Managementebene eskaliert werden. Da die Eskalation im allgemeinen nur eine längerfristige Lösung ermöglicht, sollte trotzdem versucht werden, möglichst viele Workflow-SLAs einzuhalten. Genauer, der Wert der Workflows, deren SLAs eingehalten werden, soll maximiert werden. Dazu ist es nötig, jedem Workflow einen Wert zuzuweisen.

Definition 3.4.1. Jedem Workflow w wird ein bestimmter Betrag als Wert $\mathfrak{v}_w$ zugewiesen. Für einen bestimmten Zeitpunkt t ist der Gesamtwert $\mathfrak{v}_{\mathfrak{S}}$ des Systems die Summe der Werte aller Workflows, deren SLA eingehalten wurde.

In einem realen System wird ein Workflow anhand verschiedener Kriterien bewertet, beispielsweise anhand tatsächlicher monetärer Erträge die durch die Erfüllung eines SLAs dem Unternehmen zu gute kommen. Dies erfolgt anhand bestimmter Prozesse, wie dem Budgeting und dem Accounting der IT Infrastructure Library [Ogc01]. Der Wert kann wie ein SLA auf die verschiedenen Aktivitäten aufgeteilt werden, d.h. welchen Beitrag die einzelnen Aktivitäten zum Gesamtwert des Workflows beitragen.

Die Nichteinhaltung eines SLAs kann mit Strafzahlungen belegt sein. Diese sollten im Wert des Workflows widergespiegelt werden. Beispielsweise sollte der Wert eines Workflows erhöht sein, wenn eine SLA Verletzung höhere Strafzahlungen nach sich zieht, d.h. der Wert wird so normalisiert, dass bei Nichteinhaltung des SLAs der Wert eines Workflows genau 0 ist.

3.4.1 Lockerung des SLOs

Als erstes wird die Lockerung eines SLOs betrachtet. Es geht also darum, das SLO soweit zu erhöhen, dass dieses größer als die tatsächlich gemessene Antwortzeit ist. Ein SLO, das einer Aktivität a zugeordnet ist, kann nur gelockert werden, wenn die SLOs aller Aktivitäten eines Workflows anschließend weiterhin den Anforderungen aus Definition 3.3.2

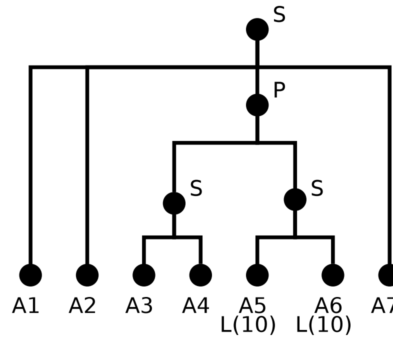

Abbildung 3.3: Die Transformation des Workflows aus Abbildung 3.2 in eine Baumstruktur.

entsprechen. Es muss deshalb auf allen Pfaden, die a enthalten, das SLO um den Betrag gestrafft werden, um den das SLO von a gelockert werden soll. Für parallel verlaufende Teilpfade ergeben sich Lockerungen der SLOs um den selben Betrag wie für Aktivität a als Seiteneffekt. Der Betrag, um den ein SLO gelockert oder gestrafft werden soll, wird im weiteren Verlauf als SLO-Anteil Δslo bezeichnet.

Um ein SLO zu lockern, müssen also folgende Schritte durchgeführt werden:

- Bestimmung von benötigten oder abzugebenden SLO-Anteilen für jede Aktivität.
- Verteilung der Informationen zwischen betroffenen Aktivitäten anhand der Struktur des Workflows.
- Ermittlung einer möglichen Lösung, die Definition 3.3.2 und 3.3.5 einhält.

Die Entscheidung über benötigte oder abzugebende SLO-Anteile muss für jede Aktivität einzeln getroffen werden. Dazu wird die Antwortzeit ermittelt (vgl. Definition 3.3.4) und mit dem SLO verglichen. Falls $\Delta slo = slo(a) - t_{response}(a) < 0$ müssen $|\Delta slo|$ SLO-Anteile hinzugewonnen werden, für $\Delta slo = slo(a) - t_{response}(a) > 0$ können SLO-Anteile abgegeben werden.

Um die Informationen aus den Entscheidungen der einzelnen Aktivitäten zu verteilen, lässt sich die Struktur des Workflows nutzen. Atomare und zusammengesetzte Aktivitäten bilden eine Baumstruktur (Abbildung 3.3) [SK08]. Jede zusammengesetzte Aktivität aggregiert die SLOs und Antwortzeiten der in ihr enthaltenen Aktivitäten.

Innerhalb einer Sequenz $\mathcal{S}$ lässt sich eine Lösung für die Verschiebung von SLO-Anteilen ermitteln, sofern für alle Aktivitäten $a \in \mathcal{S}$ einer Sequenz $\sum t_{response}(a) < \sum slo(a)$ gilt. Andernfalls muss eine zusammengesetzte Aktivität, welche diese Sequenz enthält, eine

Lösung ermitteln. Dies wird rekursiv bis zur Wurzel des Workflow-Baums durchgeführt. Wurde ein Lösung ermittelt, wird die Lösung wieder über den umgekehrten Weg an alle Kinder bis zu den atomaren Aktivitäten verteilt. Wird auch durch die Wurzel keine Lösung gefunden, wird eine SLA-Verletzung an eine Instanz außerhalb des Systems eskaliert.

Das zu lösende Problem innerhalb einer Sequenz hat folgende Eigenschaften:

- Gegeben sind die Antwortzeiten und die SLOs aller Aktivitäten der Sequenz.
- Alle Antwortzeiten als auch SLOs sind Vielfache einer kleinsten diskreten Zeiteinheit und lassen sich bis zu dieser Zeiteinheit beliebig aufteilen.

Das Problem ist folglich, vorhandene SLO-Anteile auf geforderte Antwortzeiten aufzuteilen. Dieses Problem ist in zwei Schritten zentral zu lösen, sofern eine Lösung existiert, da sich SLO-Anteile beliebig bis auf die kleinste Zeiteinheit aufteilen lassen:

1. Bildung der Summe aller SLOs aller Aktivitäten.
2. Zuteilung von SLOs an jede Aktivität, so dass die geforderte Antwortzeit eingehalten wird.

Nach der Zuteilung der SLOs können weitere SLO-Anteile übrig sein, sofern die Summe aller Antwortzeiten kleiner als die Summe aller SLOs ist. Diese Anteile werden auf alle Aktivitäten einer Sequenz aufgeteilt.

Der dargestellte zentrale Algorithmus besitzt trotz offensichtlicher Effizienz einige negative Eigenschaften:

- Alle Aktivitäten müssen alle private Informationen preisgeben.
- Alle Aktivitäten müssen sich möglicherweise der Entscheidungen von Aktivitäten außerhalb der eigenen administrativen Domäne beugen.

Da dies in einem dezentralen System, das sich über unterschiedliche administrative Domänen erstreckt, nicht aufrecht zu erhaltende Eigenschaften sind, soll im Folgenden ein dezentraler Algorithmus gefunden werden, der diese Probleme umgeht.

3.4.2 Verkürzung der Antwortzeit

Die Antwortzeit kann verkürzt werden, indem einem Service zur Abarbeitung einer Aktivität mehr physikalische Ressourcen zugeordnet werden. Dafür existieren im Allgemeinen mehrere Möglichkeiten, z.B.:

- Beschaffung zusätzlicher Ressourcen aus einem Ressourcenpool [RRC+06]
- Verschiebung von Ressourcen oder Verlegung von Aktivitäten zwischen Services [ZQS05]
- Veränderung der Scheduling-Prioritäten für Aktivitäten

All diesen Methoden ist gemein, dass die dem Service zur Verfügung stehenden Ressourcen um einen bestimmten Betrag erhöht werden müssen, um die Antwortzeit auf den erforderlichen Betrag zu verringern. Der Ressourcenbedarf besteht typischerweise aus einer Kombination unterschiedlicher Ressourcen [Mar08]:

- Prozessorzeit
- Speichermenge
- Netzwerkbandbreite
- I/O-Bandbreite

Dem Managementsystem werden die Ressourcenanforderungen mitgeteilt. Dieses versucht im Anschluss den Bedarf zu erfüllen. Im Weiteren wird diskutiert, wie durch Veränderungen am Scheduling die Antwortzeiten theoretisch verändert werden können.

Erhöhung der Priorität einer Aktivität kann zur Reduktion der Antwortzeiten für diese Aktivität führen. Dazu muss für jede Ressource ein geeignetes Schedulingverfahren implementiert werden. Um Dienstgüte-Eigenschaften durchsetzen zu können, müssen die Schedulingverfahren diese Eigenschaften durchsetzen können. Für die Prozessorzeit ist beispielsweise ein Fair Share Scheduler [KL88, Tan01] geeignet, der die zur Verfügung stehende Prozessorzeit auf einzelne Nutzer aufteilt und die Einhaltung erzwingt.

Für Netzwerk- und I/O-Bandbreite sind geeignete Schedulingverfahren Weighted Fair Queueing (WFQ) und Leaky Bucket [KR02]. WFQ kann die genutzte Bandbreite für verschiedene Nutzer partitionieren, in Kombination mit einem Leaky Bucket-Algorithmus

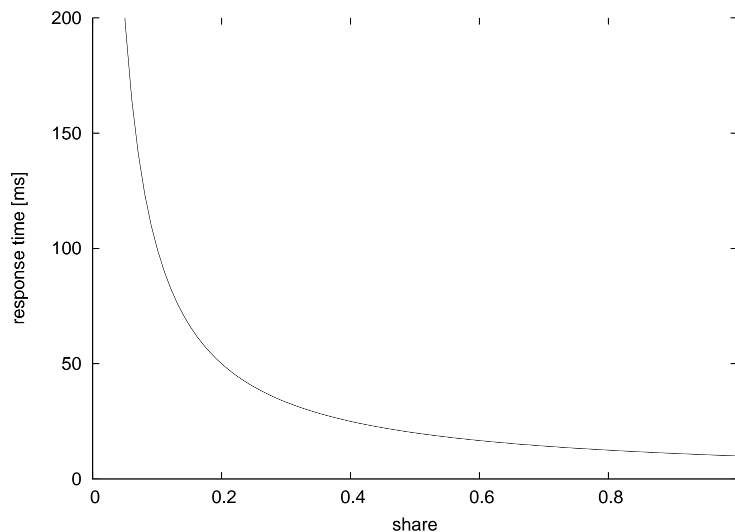

Abbildung 3.4: Scheduling-Modell eines Services mit idealer Antwortzeit von 10 ms.

kann die maximale Verzögerungszeit für einen Nutzer garantiert werden, sofern der genutzte Kommunikationspfad diese Mechanismen unterstützt [KR02].

Ein geeignetes Scheduling für die Speichermenge ist schwieriger, da eine bestimmte Speichermenge als Auslagerungsspeicher auf langsameren Speichermedien untergebracht werden kann. Dadurch entstehen große, nicht vorhersagbare Sprünge in der tatsächlichen Ausführungszeit.

Da die Nachbildung eines realen Systems in Hinblick auf die Scheduling-Eigenschaften sehr komplex ist, wird vereinfacht angenommen, dass nur ein Scheduling über Anteile an der Antwortzeit durchgeführt wird. Die ideale Antwortzeit des Services wird auf die einzelnen Aktivitäten aufgeteilt, so dass für jede einzelne Aktivität die Antwortzeit länger wird:

Die Antwortzeiten einer Aktivität a lassen sich wie folgt berechnen:

$$t_{response}(a) = \frac{t_{response}}{share(a)},$$

wobei $t_{response}$ die ideale Antwortzeit des Service und $share(a)$ der Anteil der Aktivität a ist.

Für $share(a) \to 1$ strebt die Antwortzeit für die Aktivität a gegen die ideale Antwortzeit des Service $t_{response}$, für $share(a) \to 0$ strebt die Antwortzeit gegen ∞ (vgl. Abbildung 3.4). Es lässt sich auf diese Weise leicht berechnen, wieviele Anteile an der Antwortzeit abgegeben werden können oder wieviele Anteile hinzugewonnen werden müssen.

Es wird davon ausgegangen, dass mehrere Aktivitäten den selben Service nutzen (Abbildung 3.5). Für jede Aktivität kann berechnet werden, wieviele Anteile an der Antwortzeit benötigt werden, dies kann dazu genutzt werden, das Scheduling so zu verändern, dass sofern möglich, alle Aktivitäten eine Antwortzeit unterhalb ihres SLO-Zielwerts erreichen.

Ist es nicht möglich alle Aktivitäten mit genügend Antwortzeitanteilen zu versorgen, entsteht ein schwieriges Problem. Jeder Aktivität kann aus den Workflow-Wert ein bestimmter Teil-Wert zugeordnet werden, welcher für eine Einhaltung des SLOs dieser Aktivität angerechnet werden kann. Das Problem ist nun, ein Scheduling zu finden, das die Summe aller dieser Werte maximiert. Dies ist ein Knapsack-Problem [Pap93].

3.4.3 Kombination beider Verfahren.

Kombiniert man das Verschieben von SLO-Anteilen mit dem Scheduling-Verfahren ergibt sich ein komplexeres Problem. Da SLO-Anteile in einer Sequenz verschoben werden können, kann das Scheduling für eine andere Aktivität oder mehrere Aktivitäten verändert werden, um eine SLO-Verletzung zu beheben (Abbildung 3.6).

Für Aktivitäten, die Kapazität benötigen, gibt es nun potenziell mehrere Möglichkeiten, die benötigte Kapazität zu erhalten. Für Aktivitäten, die Kapazität abgeben können, ergeben sich ebenfalls mehrere potenzielle Möglichkeiten, die benötigte Kapazität abzugeben.

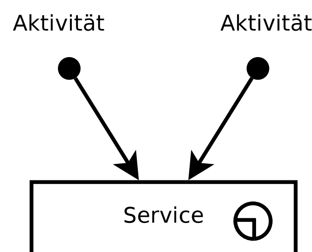

Abbildung 3.5: Zwei Aktivitäten die den gleichen Service nutzen.

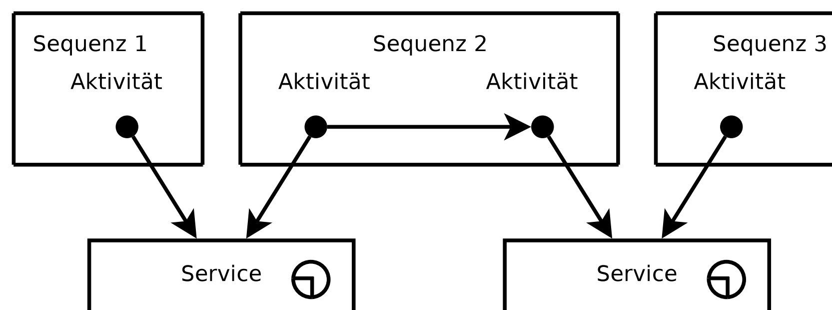

Abbildung 3.6: Sequenz 2 teilt sich zwei Services mit jeweils einer anderen Sequenz.

Aktivitäten die keine Veränderung benötigen, können außerdem als Zwischenhändler auftreten. Solche Aktivitäten verschieben Scheduling- und SLO-Anteile mit dem Ziel, Ressourcen im System besser zu verteilen.

Um gezielt Anteile verschieben zu können, müssen Informationen über aktuelle Antwortzeiten und SLOs nicht nur innerhalb eines Workflows, sondern auch zwischen Workflows ausgetauscht werden. Eine dafür gut geeignete Kommunikationsstruktur muss noch entwickelt werden.

3.5 Das SLO-Spiel

Die in Abschnitt 3.1 beschriebene Problemstellung betrachtet Services, die von unterschiedlichen administrativen Bereichen verwaltet werden und damit unabhängig von den anderen Services ihren eigenen Nutzen maximieren wollen. Dies entspricht dem grundlegenden Gedanken eines Agenten in der Spieltheorie (vgl. Abschnitt 2.3.3). Es bietet sich deshalb an, das Problem mittels Mechanismus-Design anzugehen und einen Mechanismus zu entwerfen, der das Problem näherungsweise lösen kann.

Ein Ziel ist es, SLAs garantieren zu können. Deshalb ist eine wesentliche Eigenschaft des Mechanismus, wie er sich im schlechtesten Fall verhält. Weiterhin ist es interessant, wie das Ergebnis im Vergleich zu einem optimalen Algorithmus ausfällt. Zuerst wird nur die Verschiebung von SLO-Anteilen in Sequenzen betrachtet. Anschließend wird überprüft, ob die Eigenschaften aus dem einfacheren Spiel auch in verallgemeinerter Form gelten, wenn komplette Workflows und auch Veränderungen am Scheduling mit betrachtet werden.

3.5.1 Mechanismus

Der Mechanismus wird nach dem in Abschnitt 2.3.4 definierten Modell definiert:

Definition 3.5.1 (SLO-Spiel)**.** Das SLO-Spiel ist ein wiederholter Mechanismus, der einen statischen Mechanismus $M = (\mathfrak{A}, \mathfrak{S}, O, f, p_1, \cdots, p_n)$ wiederholt ausführt. Der wiederholte Mechanismus wird unendlich oft wiederholt, d.h. es ist den Agenten nicht bekannt, wann sie das Spiel beenden.

- Die Menge der Agenten $\mathfrak{A}$ entspricht der Menge der Aktivitäten A.

- Die Menge möglicher Strategien $\mathfrak{S}$ ist die Deklaration eines Geldbetrags für einen bestimmten SLO-Anteil. Alternativ kann sich ein Agent dafür entscheiden eine Auktion durchzuführen und selbst SLO-Anteile verkaufen.
- Die Menge möglicher Ergebnisse O ist die Zuordnung von SLO-Anteile zu Agenten.
- Die Ergebnisfunktion f ist eine dezentrale verallgemeinerte Vickrey-Auktion, d.h. jeder Agent kann Anteile selbst verkaufen ohne einen zentralen Auktionator.
- Die Zahlungsfunktionen p_i sind die Zahlungsfunktionen der Vickrey-Auktion. Falls der Gewinner der Auktion der einzige Bieter ist, muss er einen vom Verkäufer festgelegten Reservierungspreis zahlen.

Die Menge der Agenten ist auf diese Weise gewählt, da jede Aktivität selbst eine Menge Geld abhängig vom jeweiligen Workflow zugewiesen bekommt und deshalb die Entscheidungen im Wesentlichen für jede Aktivität separat getroffen werden sollten. Weiterhin wird dadurch die theoretischen Eigenschaften eines direkt enthüllenden (vgl. Definition 2.3.5) und anreizkompatiblen (vgl. Definition 2.3.7) Mechanismus möglich. Da Agenten, die vom gleichen Service ausgeführt werden, prinzipbedingt über ein vollständiges Wissen übereinander besitzen, kann dies allerdings für Manipulationen einzelner Auktionen eingesetzt werden.

Effizienz

Definition 3.5.2. Ein Ergebnis $o \in O$ ist dann optimal, wenn der Wert $\mathfrak{v}$ der eingehaltenen Workflow-SLAs maximiert wird:

$$\mathfrak{v} \in \max_{o \in O} \sum_{w=1}^{n} \mathfrak{v}_w(o)$$

Wobei n die Anzahl aller Workflows ist und $\mathfrak{v}_w$ der Wert eines Workflow, falls das Workflow-SLA eingehalten wird. Andernfalls gilt $\mathfrak{v}_w = 0$.

Eine schwächeres Effizienz-Kriterium betrachtet nicht das gesamte System, sondern einzelne Sequenzen (Workflows):

Definition 3.5.3. Ein Ergebnis $o \in O$ ist dann in Bezug auf einen Sequenz $\mathcal{S}$ (oder Workflow w) effizient, wenn alle SLOs aller an $\mathcal{S}$ (w) beteiligten Aktivitäten a eingehalten

werden. Dies ist dann möglich, wenn die Sequenz (alle Pfade durch w) eine Gesamtausführungszeit kleiner dem Workflow-SLA $sla(w)$ hat:

$$\sum_{i=1}^{j} t_{response}(a_i) \leq sla(w)$$

Strategien und Wertfunktionen

Eine Strategie eines Agenten i ist das Bieten eines Typs t_i. Wenn der Mechanismus anreizkompatibel ist, ist die Gleichgewichtsstrategie das Bieten des tatsächlichen Werts. Der Wert wird durch eine Wertfunktion berechnet, die im folgenden entwickelt wird.

Ein Agent hat folgende Informationen, die für die Berechnung des Werts eines SLO-Anteils herangezogen werden können:

- Der Wert der Workflows und der SLOs.
- Das geforderte SLO und das maximal erreichbare SLO
- Die tatsächliche Antwortzeit

Die erste interessante Information für einen bietenden Agent ist, ob nach einer Auktion oder mehrerer Auktionen das SLA eingehalten wird oder nicht, da nur dann ein Ergebnis einen tatsächlichen Wert hat, wenn das SLA eingehalten wurde. Kann ein Agent feststellen, ob das SLA insgesamt eingehalten wurde, ohne vollständiges Wissen über die anderen Agenten zu besitzen? Wie sich herausstellt, ist es theoretisch möglich, dass ein Agent aus den einzelnen Auktionen Rückschlüsse ziehen kann, ob ein SLA eingehalten wurde, oder nicht:

Lemma 3.5.4. *Jeder Agent, der SLO-Anteile kaufen will, sendet eine Anfrage bezüglich laufender Auktionen an alle Agenten, mit denen er SLO-Anteile austauschen kann.*

Dann ist das SLA garantiert erfüllt, wenn ab einem bestimmten Zeitpunkt t *keine Anfragen mehr versendet werden. Weiterhin ist das SLA sicher verletzt, wenn auf eine Anfrage keine Antwort erhalten wird.*

Beweis. Ein Agent sendet immer dann eine Anfrage, wenn eine SLO-Verletzung aufgetreten ist. Wird ab Zeitpunkt t keine Anfrage mehr gestellt, halten alle Agenten ihre SLOs ein und damit wird auch das gesamte SLA eingehalten.

Der zweite Teil ist analog zum ersten. Wenn ein Agent auf eine Anfrage keine Antwort erhält, gibt es keine Agenten, dessen SLO kleiner als die tatsächliche Antwortzeit ist. Das SLA muss nun verletzt sein, da der Agent, welcher die Anfrage stellt, sein SLO sicher verletzt. □

Die Bestimmung des Zeitpunkts t ist allerdings nur möglich, wenn eine obere Schranke Δt für die benötigte Zeitdauer von der Feststellung der SLO-Verletzung bis zum Eintreffen der Anfrage bei einem anderen Agenten bekannt ist. Im umgekehrten Fall besteht das gleiche Problem, allerdings ist die benötigte Zeitspanne Δt größer, nämlich die Zeit, die maximal zwischen dem Senden von einer Anfrage und dem Eintreffen einer Antwort vergehen darf.

Weiterhin gibt es Probleme, auf diese Weise festzustellen, ob ein SLA eingehalten wurde oder nicht, sofern sich die Antwortzeiten der Agenten dynamisch und oft ändern. Es lässt sich dann weiterhin nicht feststellen ob ein SLA verletzt ist oder nicht, sofern das System nicht für mindestens die Zeitspanne Δt zu Ruhe kommt.

Es ist deshalb besser, eine Aufteilung des Werts des Workflows auf die einzelnen SLOs durchzuführen. Dies erfolgt in Form eines Geldbetrags, der einem Agenten bei seiner Erzeugung zugewiesen wird. Die Einhaltung eines SLOs kann immer direkt am Agent selbst geprüft werden durch den Vergleich von SLO und tatsächlicher Antwortzeit. Da ein Agent kein Gebot auf das Gesamt-SLO abgibt, ermittelt die Wertfunktion den Wert der nötigen SLO-Anteile, um die SLO-Verletzung zu beheben. Der Wert der SLO-Anteile wird mittels des vorhandenen Geldbetrags ausgedrückt. Dazu muss der Wert und die initial zugeteilte Geldmenge in einem Zusammenhang stehen.

Definition 3.5.5. Die Wertfunktion eines bietenden Agenten i ist folgendermaßen definiert:

- $m(i)$ ist das aktuelle Budget des Agents i.
- $slo(i)$ ist das aktuelle SLO des Agents i.
- $\overline{slo(i)} = sla(w) - (\overline{\#(p(i))} - 1)$ ist das maximal erreichbare SLO des Agents i, wobei $\overline{\#(p(i))}$ die Länge des längsten Pfads durch den Workflow w ist, in welchem i enthalten ist.
- Δslo sind die durch eine Auktion zugeteilten SLO-Anteile.

$$v_i =_{def} \begin{cases} 0, & \text{falls keine SLO-Anteile zugeteilt wurden} \\ \frac{m(i)}{\overline{slo(i)} - slo(i)} \cdot \Delta slo, & \text{sonst} \end{cases}$$

Anschaulich bewertet ein Agent also jeden Anteil bis zum maximal erreichbaren SLO gleich. Der Wert der Anteile steigt mit der Menge linear an. Der Grund für diese Bewertung liegt darin, dass dadurch sichergestellt ist, dass einem Bieter niemals das Geld ausgeht. Auf Grund dieser Bewertung der SLO-Anteile über den Geldbetrag kann durch den Geldbetrag die Priorität eines Workflows ausgedrückt werden.

Für den Verkäufer nehmen wir an, dass zusätzliche, nicht genutzte SLO-Anteile prinzipiell wertlos sind, da diese nichts zur momentanen Erfüllung des SLOs beitragen. Dies entspricht nicht unbedingt der Realität, beispielsweise könnte einem Agent bekannt sein, dass die Auslastung häufig schwankt und die zusätzlichen SLO-Anteile immer wieder in Anspruch genommen werden, der Wert der SLO-Anteile wäre in diesem Fall $v_i > 0$.

Definition 3.5.6. Die Wertfunktion eines Agents i der sein SLO einhält, also auch für Verkäufer, ergibt immer der Wert des aktuellen SLOs, dieser ist abhängig von der aktuellen Antwortzeit und dem vorhandenen Geldbetrag. Dabei gilt:

- $m(i)$ ist das aktuelle Budget des Agents i.
- $\overline{slo(i)} = sla(w) - (\overline{\#(p(i))} - 1)$ ist das maximale SLO des Agents i, wobei $\overline{\#(p(i))}$ die Anzahl der Agenten ist, die im längsten Pfad durch den Workflow w enthalten sind, in welchem auch i enthalten ist.
- $t_{response}(i)$ ist die aktuelle Antwortzeit, die für eine Aktivität ermittelt wurde.

$$v_i =_{def} \frac{m(i)}{\overline{slo(i)}} \cdot t_{response}(i)$$

Wiederholtes SLO-Spiel

Das SLO-Spiel ist ein wiederholtes Spiel. Das Problem ist hier, dass, wie in Abschnitt 2.3.9 beschrieben, Auktionen im wiederholten Fall den Nachteil haben, teilweise nur ein sehr geringes Einkommen für den Verkäufer zu erzielen, sofern die Agenten genügend Geduld aufweisen. Um das gewünschte Verhalten für $\delta \rightarrow 1$ zu erzwingen, muss eine geeignete Strafe entwickelt werden, welche die Agenten dazu zwingt möglichst früh am SLO-Spiel teilzunehmen.

Vollständig geduldige Agenten können solange warten, bis sie der einzige Agent in der Auktion wären, d.h. der zu zahlende Preis wäre immer 0, bzw. der Reservierungspreis. Um einen geduldigen Bieter dazu anzureizen früher zu bieten, müsste bei einem nicht erfolgten Gebot der zukünftige Nutzen ähnlich einer Minimax-Strategie niedriger ausfallen als bei sofortigem Bieten. Eine solche Strategie, die ohne vollständiges Wissen auskommt, ist nicht zwangsläufig tatsächlich vorhanden [GC07].

In einem realen System führt eine hohe Anzahl von SLO-Verletzungen automatisch zu einem dauerhaften Verlust, da der Vertrag zwischen dem Service-Anbieter und dem Kunden irgendwann gekündigt wird und dadurch keine Einnahmen mehr erzielt werden. Deshalb wird im hier vorgestellten SLO-Spiel vorausgesetzt, dass die Agenten ungeduldig sind, da eine zu häufige Anzahl von SLO-Verletzungen zu einer drastischen Reduktion des Nutzens führt. Ungeduldige Agenten werden die Vickrey-Auktion wie gewohnt durchführen und damit immer das dominante Gleichgewicht wählen.

Strategien zur Einkommensmaximierung

Problematisch ist, das durch die Vickrey-Auktion die Einnahmen des Verkäufers ohne Reservierungspreis sehr niedrig ausfallen können. Ohne Reservierungspreis sind die Einnahmen im schlechtesten Fall 0. Dieser Fall tritt immer dann auf, wenn nur ein Agent an der Auktion beteiligt ist.

Für den Verkäufer ist es allerdings nötig, eine Strategie für einen möglichst hohen Reservierungspreis zu finden.

- Das Einkommen für einen Verkäufer sollte so nah wie möglich an dem Wert liegen, den ein Käufer einem SLO-Anteil zumisst.

Zuerst betrachten wir den Verkäufer i und wie dieser den Reservierungspreis r_i gestalten kann. Der Verkäufer bewertet den zu verkaufenden SLO-Anteil ähnlich wie der Bieter und setzt diese Bewertung als Reservierungspreis. Bewertet wird tatsächlich die kleinste diskrete Einheit, in die SLO-Anteile zerteilt werden können (vgl. Abschnitt 3.3).

$$r_i(1) = \frac{m(i)}{\overline{slo(i)} - slo(i)}$$

Anschließend wird der Reservierungspreis solange um einen Betrag Δr reduziert, bis alle SLO-Anteile verkauft wurden, wobei:

$$r_i(t) = r_i(t-1) - \Delta r$$

Sobald $r_i(t) < 1$ wird die Auktion nicht mehr angeboten und als gescheitert erklärt.

Satz 3.5.7. *Der minimale Ertrag eines Verkäufers, wenn er alle SLO-Anteile verkauft hat, ist*

$$e_i = (r_i(1) - \Delta r \cdot a) \cdot \Delta slo,$$

wobei a die Anzahl Auktionen ist, die benötigt wurden um alle SLO-Anteile Δslo zu verkaufen. Bieten die Bieter wahrheitsgemäß, ist dies gleichzeitig eine untere Schranke des Ertrags.

Beweis. Es gibt maximal

$$\overline{a} = \frac{r_i(1)}{\Delta r}$$

Auktionen, bis die Auktion ausgesetzt wird, da die Auktion als erfolglos abgebrochen wird, wenn $r_i(t) = 1$ erreicht wurde.

Wurden schon nach $a \leq \overline{a}$ Auktionen alle SLO-Anteile verkauft gilt:

$$e_i = (r_i(1) - \Delta r \cdot a) \cdot \Delta slo,$$

da spätestens in der a-ten Auktion alle SLO-Anteile Δslo mindestens zum Reservierungspreis $r_i(a) \geq 1$ verkauft wurden. □

Verallgemeinerte Vickrey-Auktion mit Reservierungspreis

Die Vickrey-Auktion für einzelne Objekte kann verallgemeinert werden, so dass mehrere SLO-Anteile in einer Auktion an unterschiedliche Bieter vergeben werden können [AC04].

Definition 3.5.8 (Verallgemeinerte Vickrey-Auktion)**.** Die verallgemeinerte Vickrey-Auktion ist eine Auktion, die eine Menge von diskreten, äquivalenten Objekten $\mathfrak{O}$ versteigert. Jeder Bieter i gibt für eine Menge von Objekt $O_i \in \mathfrak{O}$ ein Gebot ab, das aus dem Gebot b_i für ein Objekt besteht. Weiterhin legt der Verkäufer der Objekte einen Reservierungspreis p_r fest, den jeder Bieter für jedes Objekt mindestens bieten muss.

Die Ermittlung des Ergebnisses erfolgt so, dass der soziale Nutzen maximiert wird:

$$f(v_1, \cdots, v_n) \in max_{o \in O} \sum_{i=1}^{n} v_i(o)$$

Die Ermittlung von Preisen wird wieder mit der Clarke-Pivot-Regel durchgeführt. Der Reservierungspreis kann dabei als Gebot auf alle Objekte betrachtet werden. Der zu zahlende Preis p_i lässt sich wie folgt berechnen:

$$p_i = \max_{c \in O} \sum_{j \neq i} v_j(c, t_j) - \sum_{j \neq i} v_j(o, t_j)$$

Lemma 3.5.9. *Die verallgemeinerte Vickrey-Auktion ist individuell rational, anreizkompatibel und effizient.*

Beweis. Die verallgemeinerte Vickrey-Auktion ist ein VCG-Mechanismus mit der Clarke-Pivot-Regel. □

Beispiel 3.5.10. Es bieten zwei Agenten auf drei Objekte. Agent 1 bietet $\{10, 10\}$, Agent 2 bietet $\{7\}$, der Reservierungspreis beträgt $p_r = 6$. Die verallgemeinerte Vickrey Auktion ermittelt nun das Ergebnis, das zwei Objekte an Agent 1 übertragen werden sollen und eins an Agent 2. Beide Agenten zahlen den Preis $p_r = 6$ für jeweils ein einzelnes Objekt, d.h. Agent 1 zahlt 12 und Agent 2 zahlt 6.

Prinzipiell gibt es drei Gebote $\{10, 10, 7\}$, zwei Gebote von Agent 1. Betrachten wir zunächst den Preis für Agent 1 ohne Reservierungspreis:

$$\begin{aligned} \max_{b \in O} \sum_{j \neq 1} v_j(b, t_j) &= 7 \\ \sum_{j \neq 1} v_j(o, t_j) &= 7 \\ p_1 = 7 - 7 &= 0 \end{aligned}$$

Die Bezahlung des Agenten 1 wäre ohne den Reservierungspreis 0. Da der Reservierungspreis als weiteres Gebot interpretiert wird, entsteht ein höherer Wert ohne Agent 1:

$$\begin{aligned} \max_{b \in O} \sum_{j \neq 1} v_j(b, t_j) &= 19 \\ \sum_{j \neq 1} v_j(o, t_j) &= 7 \\ p_1 = 19 - 7 &= 12 \end{aligned}$$

Agent 1 zahlt in diesem Fall 12. Für Agent 2 lässt sich die gleiche Berechnung anstellen.

Ablauf des SLO-Spiels

Jede Runde des SLO-Spiels wird von einem Agenten a wie folgt durchgeführt. Jeder Agent i tätigt in jeder Runde des SLO-Spiels folgende Aktionen:

1. Bestimmung der Antwortzeit $t_{response}(a)$ und Vergleich mit SLO $slo(a)$.
2. Ein Agent i wird Verkäufer, falls $t_{response}(a) < slo(a)$, andernfalls Bieter.
3. Jeder Verkäufer ermittelt seinen Reservierungspreis und die zu versteigernden SLO-Anteile und macht anschließend seine Auktion den Bietern bekannt. Dabei wird der Reservierungspreis und die Anzahl zu vergebender SLO-Anteile mitgeteilt.
4. Jeder Bieter gibt ein Gebot an einen Verkäufer ab.
5. Jeder Verkäufer ermittelt nach Beendigung der Auktion das Ergebnis und teilt es den jeweiligen Bietern mit.

3.5.2 Eigenschaften

Für die praktische Einsetzbarkeit des SLO-Spiels sind bestimmte Eigenschaften wünschenswert. Darunter sind Budget-Ausgeglichenheit, individuelle Rationalität und Anreizkompatibilität. Weiterhin sollte das Ergebnis zu nahe wie möglich am Optimum sein und die Situation im Vergleich des Systems ohne den Mechanismus nicht verschlechtern.

Lemma 3.5.11. *Das SLO-Spiel ist budget-ausgeglichen.*

Beweis. Die Agenten führen verteilte Vickrey-Auktionen durch, bei denen die Zahlungen vom Käufer an den Verkäufer weitergeleitet werden. Dabei erfolgt weder ein Zufluss noch Abfluss von Geld aus dem System, sofern kein Agent zum System hinzugefügt oder entfernt wird. □

Das SLO-Spiel ist also budget-ausgeglichen, da sich die Anzahl der Agenten, solange das System besteht, nicht ändert. Diese Annahme besteht in einem realen System nicht. Im weiteren Verlauf wird diese Annahme beibehalten. Ob die weiteren Eigenschaften des SLO-Spiels durch das Aufheben dieser Annahme sich ändern, ist offen.

Lemma 3.5.12. *Das SLO-Spiel ist individuell rational.*

Beweis. Da jede einzelne verallgemeinerte Vickrey-Auktion individuell rational ist, sind auch wiederholt durchgeführte Auktionen individuell rational. Dies folgt direkt aus der Definition eines wiederholten Mechanismus. Das SLO-Spiel ist auch für Verkäufer individuell rational, da der Wert eines Verkäufers nicht vom Ausgang der Auktion abhängt. □

Lemma 3.5.13. *Das SLO-Spiel ist für ungeduldige Käufer und Verkäufer anreizkompatibel.*

Beweis. Da ungeduldige Bieter sofort bieten werden, wird das dominante Gleichgewicht der verallgemeinerten Vickrey-Auktion gespielt, dieses ist immer anreizkompatibel.

Verkäufer sind genauso wie Bieter ungeduldig. Da ein Verkäufer keinen Wert aus zu versteigernden SLO-Anteilen zieht, ist für ihn die Wahrheit ebenfalls eine dominante Strategie und damit anreizkompatibel. □

Satz 3.5.14. *Das SLO-Spiel erreicht nach einer endlichen Anzahl von Auktionen immer ein nach Definition 3.5.3 effizientes Ergebnis, sofern alle Pfade p durch den Workflow w eine Gesamtausführungszeit kleiner als die vorgegebene Ausführungszeit des Workflow-SLAs haben.*

Beweis. Die verallgemeinerte Vickrey-Auktion maximiert das soziale Wohl, allerdings kann das Ergebnis mit Reservierungspreis zunächst ineffizient sein. Da der Verkäufer den Reservierungspreis bei jeder weiteren Auktion senkt, wird nach einer endlichen Anzahl von Auktionen allerdings ein effizientes Ergebnis erreicht, da es immer einen Punkt gibt, an dem ein Bieter den Zuschlag bekommen kann, da den Bietern nie das Geld ausgeht, solange das SLO kleiner dem maximal erreichbaren SLO eines Agenten ist. □

Lemma 3.5.15. *Manipulation der Preise führt zu Verzögerungen, aber kann nicht verhindern, dass nach einer endlichen Anzahl Schritten ein effizientes Ergebnis erreicht wird.*

Beweis. Es gibt zwei mögliche Manipulationen:

- Der Verkäufer lügt bei der Angabe der zu zahlenden Beträge und erhöht damit den Verkaufspreis.
- Die Käufer senken den Verkaufspreis durch Absprachen.

Fall 1

Der Verkäufer lügt. Dies hat für den einzelnen Agenten die gleiche Auswirkung wie ein

höherer Reservierungspreis, d.h. er zahlt mehr, die Zuweisung bleibt allerdings weiterhin effizient. In späteren Auktionen besitzt der zu viel zahlende Bieter weniger Geld.

Fall 2
Die Käufer senken den Verkaufspreis durch Absprachen. Dies hat prinzipiell die selben Folgen, wie die Manipulation durch den Verkäufer, allerdings ist diesmal der Verkäufer betroffen, der im späteren Verlauf des SLO-Spiels weniger Geld zur Verfügung hat.

Da jeder Käufer nur eine bestimmte Geldmenge maximal ausgibt, um SLO-Anteile zu kaufen und ein Verkäufer mindestens eine Geldeinheit pro SLO-Anteil als Reservierungspreis festlegt, wird das SLO-Spiel trotzdem zu einem effizienten Ergebnis nach Definition 3.5.3 führen. Durch niedrigere Geldmengen kann das Erreichen der effizienten Lösung allerdings verzögert werden. □

3.5.3 Komplexität

Hier werden die in Abschnitt 2.3.7 beschriebenen vier Teilprobleme betrachtet.

Eine Vorbedingung für alle Beweise ist, dass keine "großen" Zahlen genutzt werden, d.h. es werden nur ganzzahlige Werte und Fließkommazahlen endlicher Genauigkeit genutzt, die ein bestimmter Zielrechner nativ, d.h. in konstanter Zeit, addieren, subtrahieren, multiplizieren und dividieren kann.

Lemma 3.5.16. *Die Berechnung des eigenen Werts ist in $O(1)$, d.h. konstanter Zeit, möglich.*

Beweis. Die Eingabelänge für die Wertfunktion ist konstant. Die Berechnungsvorschrift ist immer identisch, somit ist die Berechnung in konstanter Zeit möglich. □

Die Berechnung und Modellierung der Strategien anderer Agenten ist nicht nötig. Der einzige Einfluss beschränkt sich auf die Beobachtung des Ausgangs einer Auktion.

Lemma 3.5.17. *Die Ergebnisfindung einer Auktion ist in $O(n \log n)$ möglich, wobei n die Anzahl der eingegangenen Gebote ist.*

Beweis. Zuerst muss eine Sortierung der Gebote erfolgen, dies ist in $O(n \log n)$ Schritten möglich.

Die restliche Berechnung beschränkt sich auf $m \leq n$ Berechnungen für das SLO, wobei m die Anzahl Zuteilungen durch die Auktion ist. slo_q sind die zur Verfügung stehenden

SLO-Anteile zu Beginn der Berechnung und slo_{q+1} sind die für die nächste Berechnung zur Verfügung stehenden SLO-Anteile. $slo_{i,wunsch}$ entspricht den im Gebot eines Agenten i gewünschten SLO-Anteilen.

$$slo_{i,zuteilung} = \begin{cases} slo_{i,wunsch}, & \text{falls } slo_q > slo_{i,wunsch} \\ slo_q, & \text{sonst} \end{cases}$$

$$slo_{q+1} = slo_q - slo_{i,zuteilung}$$

und m Berechnungen für den Preis gemäß der verallgemeinerten Vickrey-Auktion (siehe Definition 3.5.8).

Alle Berechnungen außer der Sortierung sind in $O(m) \leq O(n)$ Schritten durchführbar. □

Die Komplexität der Kommunikation ist in mehreren Schritten zu ermitteln. Zuerst wird die Kommunikation ohne Multicast per Gruppenkommunikation untersucht. Es wird von n Agenten insgesamt, b Bietern und s Verkäufern ausgegangen. Für jede Auktion gilt folglich:

- Der erste Schritt ist die Benachrichtigung der Bieter über eine durchgeführte Auktion. Dies lässt sich auf zwei Arten lösen. Entweder der Verkäufer benachrichtigt alle Agenten über die Auktion, oder jeder Bieter befragt alle Agenten.

  Für den ersten Fall müssen $n \cdot s$ Nachrichten versendet werden. Jeder Verkäufer benachrichtigt jeden Agenten. Der Nachteil ist hier, dass auch Nachrichten versendet werden, wenn keine Bieter vorhanden sind, da einem Verkäufer die Anzahl der Bieter nicht bekannt ist. Bei längerer Einhaltung aller SLOs entsteht hierdurch ein unnötiger Overhead.

  Für den umgekehrten Fall müssen $n \cdot b + s \cdot b$ Nachrichten versendet werden. Jeder Bieter sendet an jeden Agenten eine Anfrage. Die Verkäufer beantworten alle Anfragen mit einer einzigen Nachricht an den Bieter. Der Overhead ist höher als bei der ersten Variante, sobald gilt:

$$n \cdot b + s \cdot b > n \cdot s$$
$$b > \frac{n \cdot s}{n + s}$$

  Solange der Anteil der Bieter niedrig ist, ist die zweite Variante von Vorteil.

- Anschließend geben die Bieter ein Gebot ab und die Verkäufer schicken jedem Bieter eine Antwort, d.h. es werden nocheinmal $2 \cdot b$ Nachrichten versendet.

Nachrichten die an alle Agenten versendet werden, können per Multicast über einen Gruppenkommunikationsdienst versendet werden. Dies kann die Kosten je nach Struktur des Netzwerks erheblich reduzieren. Für den ersten Fall, Verkäufer benachrichtigen Bieter, werden also:

- s Multicast-Nachrichten versendet.

Für den zweiten Fall, Bieter befragen Verkäufer, werden:

- b Multicast-Nachrichten versendet und
- $s \cdot b$ Unicast-Nachrichten.

Da alle Teilprobleme in polynomieller Zeit gelöst werden können, ist auch der gesamte Mechanismus in polynomieller Zeit lösbar.

3.5.4 Verallgemeinertes SLO-Spiel

Das bisherige Spiel hat sich auf die Eigenschaften des SLO-Spiels konzentriert, in welchem ausschließlich SLO-Teile in einer Sequenz ausgetauscht werden. Es gibt zwei mögliche Verallgemeinerungen. Zum einen eine Betrachtung kompletter Workflows, zum anderen die Miteinbeziehung von Veränderungen am Scheduling.

Das SLO-Spiel mit Workflows

Betrachten wir zuerst komplette Workflows. Hier wird davon ausgegangen, dass sich ein Workflow gemäß der in Abschnitt 3.4.1 gezeigten Hierarchie in eine Baumstruktur umwandeln lässt.

Es gibt hier im wesentlichen zwei Probleme:

- Wird ein Agent rational entscheiden, Vertreter einer Sequenz oder Parallelausführung zu sein?
- Wie geht der Vertreter mit Geld sowie SLO-Anteilen um, die er kauft oder verkauft.

Im momentanen Modell wird davon ausgegangen, dass die Kosten für den Vertreter vernachlässigbar sind, d.h. es kostet den Agenten nichts, bringt aber einen gewissen Nutzen für den Agent selbst und für alle anderen Agenten. Dadurch ist die Vertreter-Rolle selbst individuell rational.

Der Umgang mit Geld und SLO-Anteilen kann auf zwei Arten erfolgen:

- Es wird das komplette Budget aller Agenten beim Kauf summiert. Beim Verkauf wird der Erlös gleichmäßig aufgeteilt.
- Es wird beim Kauf von SLO-Anteilen nur das Budget des Agenten genutzt, für den SLO-Anteile gekauft werden. Beim Verkauf von SLO-Anteilen wird es ebenso gehandhabt, d.h. werden von einer Parallelausführung SLO-Anteile verkauft, bekommen alle an der Parallelausführung beteiligten Agenten, von denen SLO-Anteile verkauft wurden, einen Anteil am Gesamterlös.

Die erste Möglichkeit ist offensichtlich nicht individuell rational für die Agenten, die vertreten werden, da SLO-Anteile bezahlt werden, die anderen Agenten zukommen. Die zweite Möglichkeit ist ebenfalls problematisch. Während der Verkauf von SLO-Anteilen individuell rational ist, da hier an alle beteiligten Agent ein entsprechender Betrag bezahlt wird.

Beim Kauf von SLO-Anteilen verhält es sich allerdings anders, da hier durch eine Parallelausführung jeder darin enthaltene Teilpfad von diesem Kauf zusätzliche SLO-Anteile erhält. Da die SLO-Anteile von anderen Teilpfaden nicht gebraucht werden, muss der einzelne Agent für die anderen Agenten mitbezahlen, da es für diese Agenten nicht individuell rational wäre, sich daran zu beteiligen. Da kein Agent der Teilpfade sich beteiligt, sind die Kosten für die SLO-Anteile höher als der Wert des SLOs. Damit ist die zweite Möglichkeit für keinen individuell rational.

Eine Alternative wäre, jeden Pfad eines Workflows separat zu betrachten. Da allerdings bei der Ausführung einer Aktivität nicht bekannt ist, welcher Pfad genutzt wird, könnte immer nur das kleinste SLO angewendet werden. Weiterhin müsste der Kauf von SLO-Anteilen innerhalb einer Parallelausführung immer von allen Teilpfaden erfolgen. Dies ist problematisch, da nicht garantiert ist, dass ein Agent bei allen Teilpfaden einen Zuschlag erhält.

Im weiteren Verlauf wird wegen der Einfachheit die *erste* Möglichkeit gewählt, trotz der offensichtlichen Eigenschaft, *nicht individuell rational* und deshalb auch *nicht anreizkompatibel* zu sein, da alle bisher bekannten Alternativen ähnliche Schwächen enthalten.

Frage, ob die Eigenschaft der Effizienz weiter bestehen bleibt.

Lemma 3.5.18. *Die Erweiterung um komplette Workflows beeinflusst die Effizienz nicht.*

Beweis. Es werden nur in Sequenzen Auktionen angeboten, d.h. die Effizienz innerhalb einer Sequenz ist nach Lemma 3.5.4 ebenso gegeben.

Da ein Vertreter nach den gleichen Vorschriften wie eine atomare Aktivität handelt, wird der Vertreter immer nur so viel Geld ausgeben, dass er SLO-Anteile bis zum für alle vertretenen Agenten maximal erreichbaren SLO kaufen kann. Damit ist auch hier gewährleistet, dass nach einer endlichen Anzahl von Auktionen ein nach Definition 3.5.3 effizientes Ergebnis erreicht wird. □

Komplexitätsbetrachtung der Erweiterung Die Erweiterung hat eine erhöhte Komplexität zur Folge:

- Zur Ermittlung des eigenen Typs, müssen zuerst alle Antwortzeiten, SLOs und Geldbeträge aller Agenten ermittelt und je nach Art des Vertreters geeignet zusammengefasst werden, d.h. es müssen $O(n)$ Rechenschritte durchgeführt werden, wobei n die Anzahl der vertretenen Agenten ist.
- Eine direkte Folge ist eine erhöhte Kommunikationskomplexität:
  - Zuerst müssen von jedem Agenten Antwortzeit, SLO und Geldbeträge abgefragt werden, d.h. es sind $2 \cdot n$ Nachrichten zu verschicken. Eine zur Anfrage an jeden vertretenen Agenten und eine Antwort an den Vertreter.
  - Danach müssen die Ergebnisse an alle Agenten verteilt werden, d.h. erneut n Nachrichten.

Die Anfrage an die Agenten ist die einzige Operation, die als Multicast per Gruppenkommunikationsprotokoll erfolgen kann, da die Verteilung der Ergebnisse sich individuell unterscheiden kann.

Das SLO-Spiel mit Veränderungen des Scheduling

Veränderungen am Scheduling erweitern das Modell um die Möglichkeit, nicht nur innerhalb eines Workflows, sondern auch zwischen Workflows eine Optimierung zu erreichen. Die erste wesentliche Frage ist, wie diese Erweiterung in das bisherige Schema eingepasst werden kann.

Die offensichtlichste Möglichkeit ist, Agenten auf dem selben Service untereinander an Auktionen teilhaben zu lassen. Agenten mit einem entsprechend höheren Budget können somit gemäß ihrer gewünschten höheren Gewichtung vom anderen Workflow Ressourcen abkaufen. Es ist interessant, ob weiterhin einzelne Workflows garantiert ihr SLA einhalten können, oder ob die bisherige Eigenschaft der Effizienz verloren geht.

Satz 3.5.19. *Das um Veränderungen des Schedulings erweiterte SLO-Spiel erreicht nicht garantiert die selbe Effizienz des einfacheren SLO-Spiels*

Beweis durch Gegenbeispiel. Angenommen es gibt zwei Workflows w_1 und w_2 die sich einen Service s teilen. w_1 ist der wichtigere Workflow und deshalb haben Agenten von w_1 ein deutlich höheres Budget als Agenten von w_2.

Weiterhin wird davon ausgegangen, dass das System nicht genug Ressourcen hat, um die SLAs von w_1 und w_2 gleichzeitig einzuhalten. Es besteht die Annahme, dass nur das SLA von w_2 eingehalten werden kann. Außerdem hat ein Agent $a_2 \in w_2$, der durch s ausgeführt wird, als einziger Agent zu verkaufende SLO-Anteile.

Ein weiterer Agent $a_1 \in w_1$ besitzt genügend Geld, um diese SLO-Anteile von a_2 abzukaufen. Nun verletzten beide Workflows das SLA, obwohl ursprünglich Workflow w_2 theoretisch sein SLA hätte einhalten können, wenn a_1 die SLO-Anteile nicht abgekauft hätte.

Es ist somit gezeigt, dass eine einfache Erweiterung um Veränderungen am Scheduling im allgemeinen Fall nicht die selbe Effizienz garantiert wie das einfachere SLO-Spiel. □

Dieses Problem tritt selbst dann auf, wenn der Service selbst als Agent auftritt und versucht seinen eigenen Wert lokal zu optimieren, ohne über die tatsächliche Einhaltung von SLAs Bescheid zu wissen. Wissen über die Einhaltung des SLAs kann ein Service auf Grund des Ergebnisses aus Lemma 3.5.4 nur sehr eingeschränkt erlangen, ohne zusätzlich mit den anderen Knoten explizit Informationen auszutauschen.

Weiterhin lassen sich mit Hilfe der einfachen Erweiterung keine Ressourcen über einen Zwischenschritt gezielt übertragen, da dazu Informationen fehlen (vgl. Abschnitt 3.4.3). Eine interessante Entwicklung wäre es, einen Mechanismus zu entwickeln, der Informationen über SLO-Anteile im System verteilt, um so gezielt Ressourcen durch Auktionen umverteilen zu können.

Dieses Problem ist deutlich schwieriger als das einfache SLO-Spiel. Während dort durch einfache lokale private Informationen ein effizientes Ergebnis herbeigeführt werden kann, müssen bei der Erweiterung einige weitere Probleme angegangen werden. Beispielsweise

wird in vielen Modellen vereinfacht angenommen, dass sowohl die Kommunikation als auch die Berechnungen kostenfrei sind. Dies ist in der Realität nicht der Fall.

[PS04] führt deshalb eine ganze Reihe neuer Begriffe ein, mit deren Hilfe man verteilte Mechanismen besser beschreiben kann:

- Aktionen zum Nachrichten weiterleiten
- Aktionen um Informationen preis zu geben
- Aktionen um Berechnungen durchzuführen

Alle drei Arten von Aktionen müssen so entworfen sein, dass ein Agent seine eigene Situation nicht durch eine Abweichung vom vorgegebenen Algorithmus verbessern kann. Analog zur Anreizkompatibilität führt [PS04] dazu die Begriffe der Kommunikationskompatibilität und algorithmische Kompatibilität ein, d.h. ein Agent muss dazu angereizt werden Nachrichten zu versenden bzw. weiterzuleiten oder bestimmte Berechnungen durchzuführen.

Ein nach Definition 3.5.2 effizienter Mechanismus bleibt ein noch offenes Problem.

3.5.5 Offene theoretische Fragestellungen

Für weitere theoretische Arbeiten sind noch viele Fragestellungen offen:

- Die Strategien von Käufer und Verkäufer halten momentan ihre Eigenschaften nur, wenn die Agenten ungeduldig sind. Für geduldige Agenten müssen andere Strategien gefunden werden. [Par07] bietet dafür einige Ansatzpunkte.
- Für gesamte Workflows sollte ein besserer Mechanismus gefunden werden, der individuell rational und anreizkompatibel ist. Die bisherigen Lösungsansätze sind nicht individuell rational oder besitzen andere Probleme.
- Das einfache Scheduling sollte durch realistischere Annahmen ersetzt werden.
- Das SLO-Spiel mit Veränderungen am Scheduling bedarf einer genaueren Untersuchung. Um die Effizienz gegenüber dem einfachen SLO-Spiel zu erhöhen, müssen deutlich mehr Informationen zwischen den Agenten ausgetauscht werden. Es müssen dazu Anreize geschaffen werden, die es den Agenten nicht mehr ermöglichen durch eine Abweichung vom vorgegebenen Mechanismus individuelle Vorteile zu

Lasten der Effizienz zu erreichen. Die Arbeiten [PS04, Par07] bieten dafür erste Ansatzpunkte.

- Das SLO-Spiel ist in der Realität kein Spiel mit gleichzeitigen Runden und die Lebenszeit von Agenten ist begrenzt, d.h. neue Agenten werden zum System hinzukommen, alte Agenten werden das System verlassen. Diese beiden Änderungen am System könnten entscheidende Auswirkungen auf die Eigenschaften des SLO-Spiels besitzen [Afe06].
- Sowohl Agenten als auch unterlagerte Netzwerke können ausfallen. Das SLO-Spiel sollte möglichst tolerant mit Ausfällen umgehen können.
- Im bisherigen Modell gibt es keine Kontrollinstanz über Geldbeträge der Agenten, d.h. jeder Agent kann sich theoretisch unendlich viel Geld selbst beschaffen, da alle Informationen für einen Agent privat sind. Hier könnten kryptographische Verfahren angewendet werden [Nao01].

3.6 Leistungsmessung und -Bewertung

Das SLO-Spiel soll einer Leistungsmessung und -Bewertung [KS93] unterzogen werden. Im Weiteren werden die grundsätzlichen Begriffe aus [KS93] in Zusammenhang mit der hier erfolgten Umsetzung des SLO-Spiels gebracht. Prinzipiell wird das SLO-Spiel in zwei Formen umgesetzt:

- Simulation
- verteiltes System

Mit Hilfe der Simulation soll die grundsätzliche Leistung und Skalierbarkeit des SLO-Spiels bewertet werden. Anschließend sollen die Ergebnisse der Simulation in einem verteilten System, das einem tatsächlichen Objektsystem zum Großteil ähnlich ist. Aus beiden Messungen lässt sich anschließend eine Leistungsprognose ableiten, wie sich eine Umsetzung des SLO-Spiels in einem realen System verhalten wird.

Sowohl für die Simulation, als auch die Messung im verteilten System müssen folgende grundsätzlichen Eigenschaften definiert werden [KS93]:

- *Kenngrößen* beschreiben Leistungsmerkmale die für die Bewertung interessant sind.

- *Zielvorgaben* für die Kenngrößen.
- Ein *Lastmodell* beschreibt vereinfacht die Arbeitslast eines Services.
- Die genutzt *Messtechnik* beschreibt den zeitlichen Umfang, Zeitpunkt der Auswertung und die Umsetzung der Messwerterfassung.
- Weiterhin muss definiert werden, welche *Messgrößen* mit Hilfe der beschriebenen Messtechnik ermittelt werden können.
- Art der *Zeitsynchronisation* (nur im verteilten System).

Kenngrößen

Kenngrößen sollten folgende Kriterien für das SLO-Spiel beschreiben:

- Das *Zeitverhalten* beschreibt das Zeitintervall, welches zwischen Erkennung einer SLO-Verletzung und deren Behebung vergeht.
- Die *Qualität* des Ergebnisses beschreibt, ob ein Ergebnis gefunden wird (vgl. Definitionen 3.5.2 und 3.5.3).
- Die *Stabilität* beschreibt, ob es zu Schwingungen im System kommen kann.
- Der *Kommunikationsaufwand* beschreibt wieviel Nachrichten in einer bestimmten Zeiteinheit gesendet werden.
- Die *Skalierbarkeit* beschreibt, wie sich die vier vorangegangenen Kenngrößen mit steigender Systemgröße verhalten.

Zielvorgaben

Hier wird für jede Kenngröße beschrieben wann ein Messergebnis besser ist und wann schlechter.

- Das Zeitverhalten ist besser, je kürzer das Zeitintervall ist.
- Erreicht das SLO-Spiel ein nach Definition 3.5.3 effizientes Ergebnis, ist die Qualität gut. Die Qualität kann schlechter sein, wenn kein Ergebnis gefunden wird. Die Qualität kann besser sein, wenn das ermittelte Ergebnis von einem nach Definition 3.5.3 zu einem nach Definition 3.5.2 effizienten Ergebnis strebt.

- Ein Ergebnis ist stabil, wenn das SLO-Spiel zu einem Ergebnis konvergiert und ohne Änderungen der Last eines Services kein Agent zu einem Bieter wird, d.h. es tritt keine weitere SLO-Verletzung auf.

- Der Kommunikationsaufwand ist besser, je weniger Nachrichten versendet werden.

- Die Skalierbarkeit ist besser, je mehr Agenten am SLO-Spiel teilnehmen können, ohne dass eine der Kenngrößen zu einem Flaschenhals wird.

Lastmodell

Zur Messung werden zwei unterschiedliche Lastmodelle genutzt:

- Ein *stabiles* Lastmodell, legt zu Beginn der Simulation Antwortzeiten für einen Service fest und ändert diese nicht mehr. Dies kann zur Überprüfung der Effizienz-Eigenschaft des SLO-Spiels genutzt werden (vgl. 3.5.14).

- Ein *dynamisches* Lastmodell, erweitert das stabile Lastmodell um dynamische Änderungen. Dadurch kann überprüft werden, ob das SLO-Spiel auf dynamische Änderung reagiert und dynamisch auftretende SLO-Verletzungen beheben kann.

Messtechnik

Die Messtechnik spielt eine zentrale Rolle bei der Bewertung des SLO-Spiels, denn was nicht gemessen wird, kann nicht bewertet werden.

- Der zeitliche Umfang der Messung erstreckt sich über die komplette Laufzeit der Simulation bzw. des verteilten Systems.

- Es wird eine *Offline-Auswertung* genutzt, d.h. die Daten werden gesammelt und erst nach dem eigentlichen Testlauf ausgewertet.

- Die Messwerte werden durch *Tracing* erzeugt, d.h. es wird jede Aktivität (z.B. Abgabe eines Gebots) zusammen mit einem Zeitstempel in einem Trace-Log abgespeichert. Das Trace-Log wird später weiter verarbeitet, um die entsprechenden Kenngrößen zu extrahieren (z.B. Anzahl aktiver Bieter zu einem bestimmten Zeitpunkt).

Messgrößen

Während der Laufzeit sollen Messgrößen ermittelt werden:

- Start- und Endzeit eines Bieters oder Auktionators
- Start- und Endzeit einer Auktion
- Reservierungspreis einer Auktion und durch eine Auktion zu verkaufende SLO-Anteile
- Zeitpunkt der Abgabe des Gebots, tatsächlicher Gebotspreis und gewünschte SLO-Anteile
- Zeitpunkt einer Nachrichtenübermittlung

Zeitsynchronisation

Zum korrekten Ablauf des SLO-Spiels und zur korrekten Messung wird in einer verteilten Implementierung des SLO-Spiels eine globale Zeit benötigt. Die einzelnen Uhren der verteilten Rechner werden mittels des Network Time Protocols (NTP) [Mil92] synchronisiert und erreichen damit eine für das SLO-Spiel ausreichende Genauigkeit, d.h. die im lokalen Netzwerk erreichbare Abweichung ist kleiner als 1 Millisekunde [Mil89]. Für die Simulations-Umgebung ist die Zeitsynchronisation nicht nötig.

Kapitel 4

Entwurf

Im vorangegangen Kapitel wurde eine detaillierte Analyse des Problems durchgeführt und das SLO-Spiel als möglichen Lösungsansatz entwickelt. Abschließend wurden Kriterien für die Leistungsmessung- und Bewertung festgelegt, welche auf die praktische Umsetzung angewendet werden sollen.

Die Architektur (Abschnitt 4.1) und der detaillierte Entwurf (Abschnitt 4.2) dieser praktischen Umsetzung werden im Folgenden beschrieben.

4.1 Architektur

Ein Ziel der Arbeit ist die Erstellung einer Simulationsumgebung und eines verteilten Systems, mit deren Hilfe das SLO-Spiel (vgl. Abschnitt 3.5) bewertet werden. Um eine geeignete Architektur zu entwickeln, wird das in Abschnitt 3.2 definierte System zusammen mit den Dienstgüte-Eigenschaften aus Abschnitt 3.3 betrachtet. Anschließend werden Ergänzungen zu diesem System beschrieben, um dieses jeweils in der Simulationsumgebung, bzw. als verteiltes System benutzen zu können

Zunächst wird das System aus einer strukturgebenden, logischen Sicht betrachtet. Diese Sichtweise auf das System besteht aus:

- Workflows und
- Aktivitäten

Workflows besitzen eine fest vorgegebene Struktur, welche durch das SLO-Spiel zu beachten ist. Jede Aktivität wird auf einen Service abgebildet, welcher die Aktivität letztendlich ausführt. Das reale System besteht folglich ausschließlich aus *Services*.

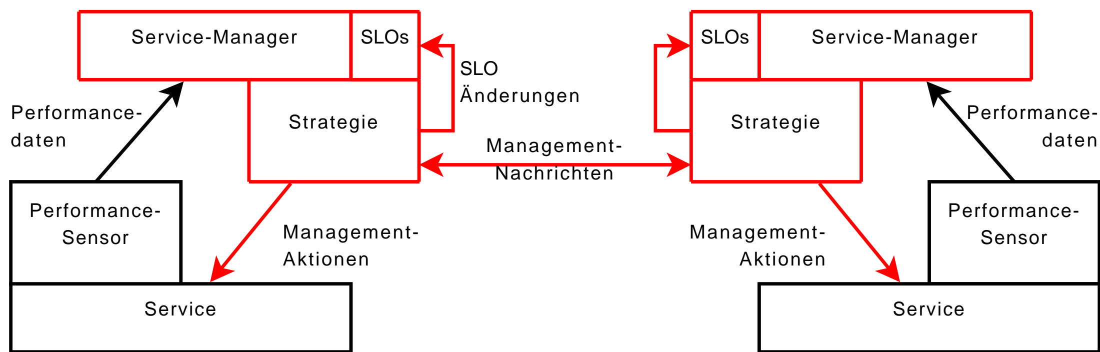

Abbildung 4.1: Die Architektur des Systems.

Die einzelnen Services sind die grundlegenden Komponenten des Systems, welche in einer realen Umgebung beispielsweise durch Webservices [ACKM04] realisiert werden.

Um die in Abschnitt 3.3 definierten Dienstgüte-Eigenschaften einhalten zu können muss jeder Service erweitert werden. Dies erfolgt in Form eines Service-Managers. Insgesamt entsteht dadurch die in Abbildung 4.1 dargestellte grundsätzliche Architektur, die im Folgenden erläutert wird.

Jeder Service wird von genau einem Service-Manager verwaltet. Ein Service muss dazu einen Performance-Sensor zur Verfügung stellen, mit dessen Hilfe der Service-Manager aktuelle Performance-Daten (für das SLO-Spiel die Antwortzeit) ermitteln kann. Außerdem kann ein Service eine weitere Schnittstelle für Management-Aktionen bereithalten. Darüber können Konfigurationsparameter geändert werden, wie beispielsweise das in Abschnitt 3.4.2 beschriebene Scheduling.

Ein Service-Manager verwaltet für jede dem Service zugewiesene Aktivität ein SLO. Das Ziel des Service-Managers ist dieses SLO durch geeignete Management-Aktionen einzuhalten.

Als Ausgangsdaten für die Management-Aktionen verwendet der Service-Manager Performance-Daten des Services und bekannte SLOs. Diese Daten werden periodisch für alle Aktivitäten ermittelt und anschließend werden daraus bestimmte Management-Aktionen initiiert. Diese Management-Aktionen werden durch eine Strategie ausgeführt.

Eine Strategie hat die Aufgabe die vom Service-Manager angestoßenen Management-Aktionen durchzuführen. Das Ziel aus der Zusammenarbeit aus Service-Manager und Strategie ist die Lösung der in Abschnitt 3.4 beschriebenen Teilprobleme.

Jede Strategie erzeugt ein Trace-Log (vgl. Abschnitt 3.6), mit dessen Hilfe man detaillierte Statistiken über den Ablauf einer Management-Aktion erzeugen kann.

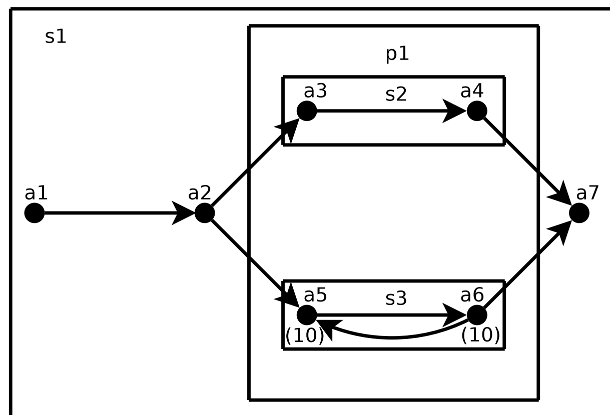

Abbildung 4.2: Gruppenbildung mehrerer Strategien.

Die Strategie kann folgende Aktionen, als SLO-Änderung oder Management-Aktion am Service (vgl. Abbildung 4.1) ausführen:

- Lockerung oder Straffung eines SLOs (vgl. Abschnitt 3.4.1)
- Beschaffung oder Abgabe von Antwortzeit-Anteilen (vg. Abschnitt 3.4.2)
- Beibehalten von SLO und Antwortzeit

Zur Übertragung von Management-Nachrichten zu anderen Strategien wird ein Gruppenkommunikationsmechanismus genutzt. Es wird für jede zusammengesetzte Aktivität, gemäß Definition 3.2.2, eine Gruppe gebildet. Der Beispielworkflow aus Abschnitt 3.2 (Abbildung 3.1) wird in die vier Gruppen $s1$, $s2$, $s3$ und $p1$ aufgeteilt (Abbildung 4.2).

Aus jeder Gruppe wird eine Strategie ausgewählt, welche in der übergeordneten zusammengesetzten Aktivität als Vertreter auftritt. Dadurch entsteht die in Abschnitt 3.4.1 dargestellte Baumstruktur des Workflows. Der Vertreter kann wie bei einer atomaren Aktivität, für die zusammengesetzte Aktivität SLOs lockern oder straffen. SLO-Anteile werden dafür mit Hilfe dieses Vertreters an andere Aktivitäten abgegeben oder von diesem Vertreter von anderen Aktivitäten an die einzelnen Aktivitäten verteilt (vgl. Abschnitt 3.5.4).

Grundsätzlich werden vom Vertreter einer zusammengesetzten Aktivität alle SLOs der Kinder zu einem virtuellen Service mit einem eigenen Service-Manager zusammengefasst. Dabei werden bei einer Sequenz alle SLOs und Antwortzeiten summiert. Bei einer Parallelausführung hat dagegen jede Kindaktivität ein gleiches SLO, allerdings eine individuell unterschiedliche Antwortzeit.

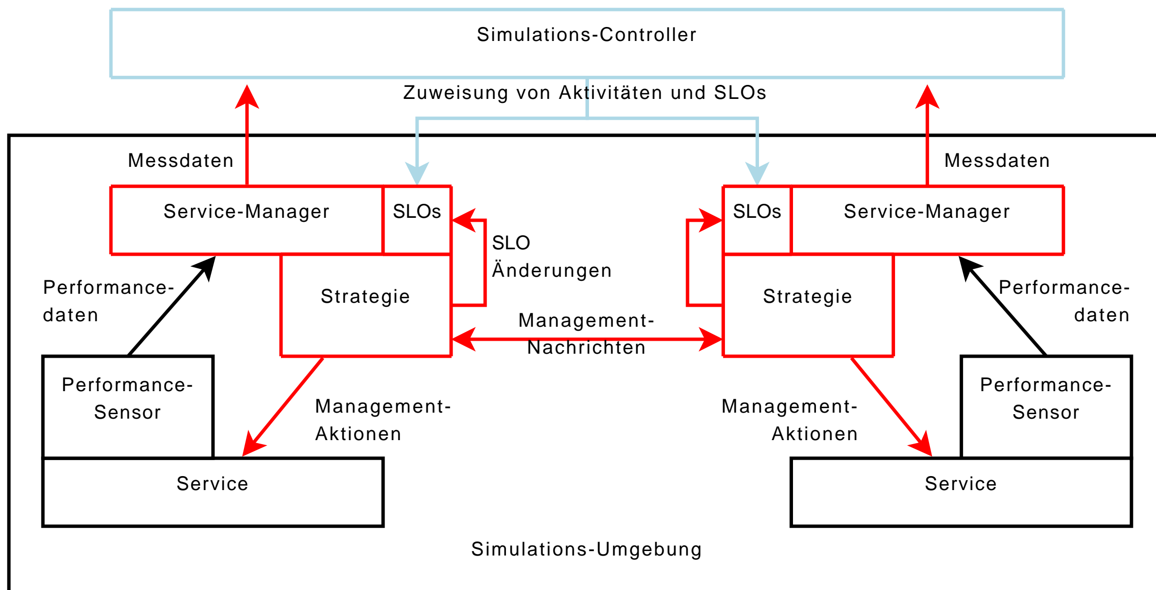

Abbildung 4.3: Die Architektur des Systems mit Simulations-Controller.

Simulation

Für die Simulation werden am bisher beschriebenen System Veränderungen vorgenommen und Komponenten hinzugefügt (Abbildung 4.3). Ergänzt wird das System durch die Einführung eines Simulations-Controllers, der für die Steuerung und Auswertung der Simulation zuständig ist. Außerdem werden alle Services und Service-Manager in der Simulations-Umgebung zusammengefasst. Simulations-Controller und Simulations-Umgebung werden durch zwei getrennte Prozesse realisiert.

Der Simulations-Controller (Abbildung 4.4) ist für die Simulation eine zentrale Kompo-

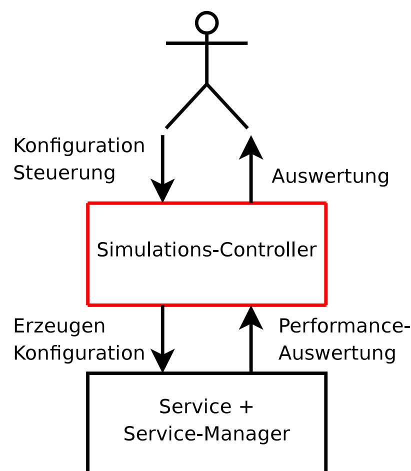

Abbildung 4.4: Die Interaktion des Simulations-Controller.

nente. Der Controller bietet eine grafische Benutzeroberfläche an, mit der die angebotenen Funktionen vom Benutzer der Simulation ausgeführt werden können. Weiterhin ist es möglich einen Überblick über den momentanen Zustand der Simulation durch eine grafische Darstellung zu erlangen.

Über die grafische Oberfläche können die folgenden Aktionen bewirkt werden:

- Erzeugen und Beenden einer Simulations-Umgebung
- Verbinden zu einer aktiven Simulation-Umgebung
- Erzeugen von simulierten Services inklusive Antwortzeitmodell
- Hinzufügen und Entfernen von Workflows
- Start und Stop eines Simulationslaufs.

Der Simulations-Controller ist von der Simulation-Umgebung getrennt, so dass sich Simulations-Controller und Simulation nicht durch Seiteneffekte beeinflussen können. Der Simulations-Controller bietet deshalb die Möglichkeit, eine neue Simulations-Umgebung zu erzeugen oder eine Simulations-Umgebung zu der er Verbunden ist zu stoppen. Durch die Trennung ist es möglich, eine Simulations-Umgebung auch ohne Simulations-Controller zu starten. Für diesen Fall bietet der Simulations-Controller die Möglichkeit, sich zu einer aktiven Simulation zu verbinden.

Sobald der Simulations-Controller mit einer Simulation verbunden ist, lassen sich Services erzeugen und Workflows hinzufügen. Bei der Erzeugung der Services wird der Simulations-Umgebung mitgeteilt wieviele Services gestartet werden sollen. Die Simulations-Umgebung erzeugt daraufhin die entsprechende Anzahl von Services und dazugehörigen Service-Managern.

Der Simulations-Controller verwaltet weiterhin ein Antwortzeitmodell für jeden Service. Anhand des Antwortzeitmodell manipuliert der Simulations-Controller die Antwortzeiten der Services und bildet auf diese Weise bestimmte durch das Antwortzeitmodell definierte Szenarien nach.

Die hinzuzufügende Workflows werden vom Simulations-Controller in ihre Aktivitäten aufgetrennt. Jede Aktivität wird anschließend zusammen mit einem aus dem Workflow-SLA abgeleiteten SLO dem zugehörigen Service-Manager zugewiesen.

Die bisherigen Aktionen beschränken sich auf die Konfiguration einer Simulations-Umgebung. Der Start eines Simulationslaufs signalisiert der Simulations-Umgebung, dass ein

Simulationslauf gestartet werden soll. Die Simulations-Umgebung beginnt daraufhin mit der Ausführung der Aktivitäten eines Service-Managers und der dazugehörigen Strategien. Der Simulations-Controller manipuliert während der Laufzeit des Simulationslaufs die Antwortzeiten der Services anhand der Vorgaben aus dem Antwortzeitmodell eines Services.

Während der Laufzeit stehen durch den Simulations-Controller zwei Möglichkeiten zur Online-Auswertung zur Verfügung:

- Gesamtübersicht
- Workflow-Details

Bei der Gesamtübersicht wird die Einhaltung von allen Workflow-SLAs gezeigt, so dass ein schneller Überblick über das gesamte System möglich ist. Unterschieden wird dabei zwischen folgenden Möglichkeiten:

- Workflow-SLA wird eingehalten (grün)
- Workflow-SLA wird eingehalten trotz vorhandener SLO-Verletzungen (gelb)
- Workflow-SLA wird verletzt (rot)

Bei den Workflow-Details werden die SLOs aller Aktivitäten eines Workflows und die Struktur des Workflows selbst visualisiert:

- Verbindungen zwischen Workflows, nach Workflow-Spezifikation
- SLO einer Aktivität wird eingehalten (grün)
- SLO einer Aktivität wird verletzt (rot)

Um dem Simulations-Controller zu ermöglichen die Antwortzeit eines Services zu manipulieren, werden die Services innerhalb einer Simulations-Umgebung durch einen simulierten Service implementiert, der als einzige Funktionalität innerhalb der Simulations-Umgebung eine Schnittstelle für Performance-Sensor und Management-Aktionen besitzt. Der Performance-Sensor selbst bezieht seine Daten aus der Antwortzeit, die über den Simulations-Controller gesetzt wurde. Aus dieser durch den Simulations-Controller zugewiesenen Antwortzeit berechnet der Service die tatsächliche über den Performance-Sensor zurückgelieferte Antwortzeit für eine Aktivität (vgl. Abschnitt 3.4.2).

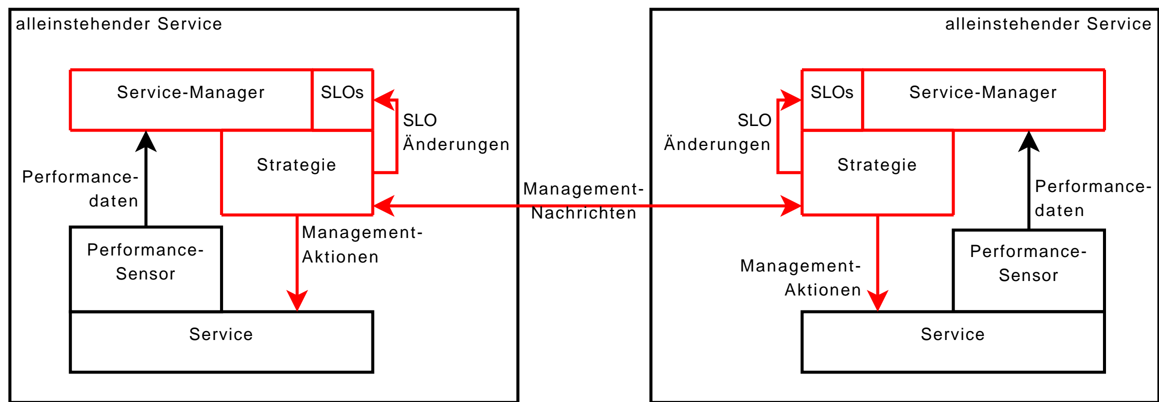

Abbildung 4.5: Die Architektur des Systems als verteiltes System.

Verteiltes System

Für die Implementierung als verteiltes System wird jeder Service als alleinstehender Service implementiert (Abbildung 4.5). Die wichtigsten Aufgaben des Simulations-Controllers übernimmt der alleinstehende Service nun selbst. Jeder alleinstehende Service wird mit den nötigen Informationen bezüglich Aktivitäten und Antwortzeitmodellen versorgt, so dass dieser sich selbst konfigurieren kann.

Für die Gruppenkommunikation der Strategien einzelner Service-Manager müssen die einzelnen Gruppenmitglieder so konfiguriert werden, dass jedem alle möglichen Gruppenmitglieder bekannt sind, da diese sich nicht zwangsläufig automatisch finden (vgl. Abschnitt 2.4.6).

4.2 Detailentwurf

Die bisherige grob umrissene Architektur der Simulationsumgebung wird nun verfeinert.

4.2.1 Service

Jeder Service des Systems muss die Schnittestelle `IService` implementieren (vgl. Abbildung 4.6). Diese Schnittstelle definiert den Performance-Sensor, um Antwortzeiten für Aktivitäten abzufragen und einen Aktor mit dem Management-Aktionen durchgeführt werden können. Für die Simulation ist der Aktor auf die Änderung von Scheduling-Prioritäten beschränkt. Weiterhin besitzt jeder Service eine global eindeutige ID, mit deren Hilfe der Service bei Management-Aktionen identifiziert wird.

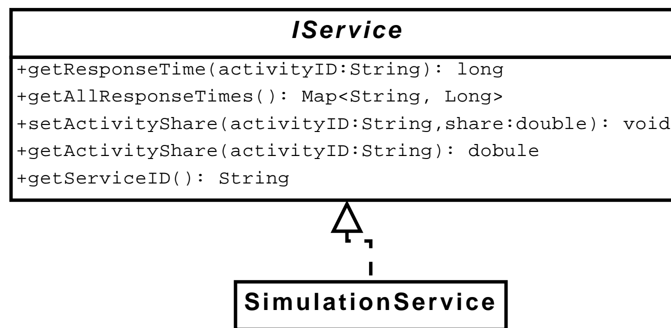

Abbildung 4.6: Schnittstelle und Implementierung des Services.

Der simulierte Service implementiert diese Schnittstelle im wesentlichen durch eine Implementierung des in Abschnitt 3.4.2 vorgestellten idealisierten Scheduling-Algorithmus. Reale Services, die in das vorgestellte System integriert werden sollen, müssen ebenfalls eine Implementierung dieser Schnittstelle zur Verfügung stellen.

4.2.2 Service-Manager

Jeder Service-Manager implementiert eine Schnittstelle `IServiceManager`, welche wiederum eine Erweiterung der `Runnable` Schnittstelle ist, d.h. jeder Service-Manager kann durch einen eigenen Thread ausgeführt werden (Abbildung 4.7).

Ein Service-Manager der Simulationsumgebung besitzt Referenzen zum verwaltenden Service, zu einem SLO-Container, welcher die SLOs enthält und zur benutzten Strategie. Daten des Services und des SLO-Containers dienen dem Service-Manager als Ausgangspunkt für Management-Aktionen, die er anschließend mittels einer Strategie durchführen

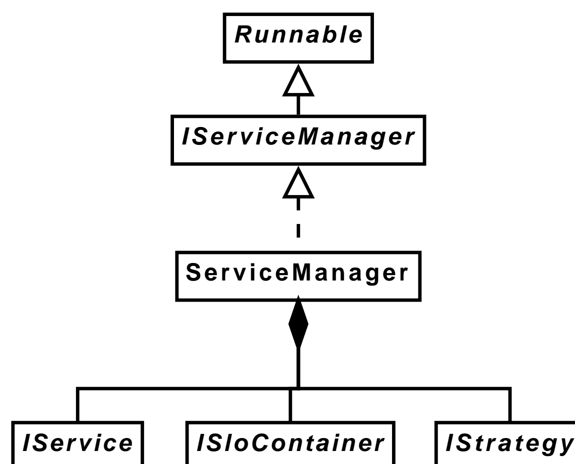

Abbildung 4.7: Klassendiagramm des Service Managers.

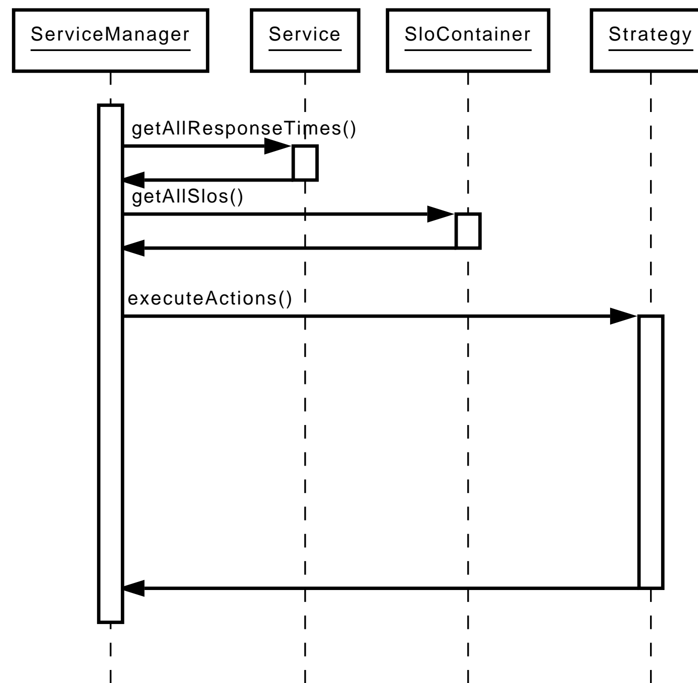

Abbildung 4.8: Der grundsätzliche Ablauf einer Management-Aktion.

lässt.

Der grundsätzliche Ablauf von Management-Aktionen wird in Abbildung 4.8 dargestellt. Zuerst werden Antwortzeiten und SLOs für alle Aktivitäten ermittelt und mit Hilfe dieser Werte von einer Strategie mögliche Management-Aktionen ermittelt und durchgeführt.

Prinzipiell wird für jeden Manager ein Thread gestartet, welcher den Service-Mager ausführt. Außerdem wird für jede Management-Aktion ein zusätzlicher Thread gestartet, da Management-Aktionen für unterschiedliche dem Service zugewiesene Aktivitäten unterschiedlich lange dauern können. Da eine große Anzahl von Service-Managern deshalb zu einer sehr großen Thread-Anzahl führen würde, ist dieser Aufbau nur mit erheblichen

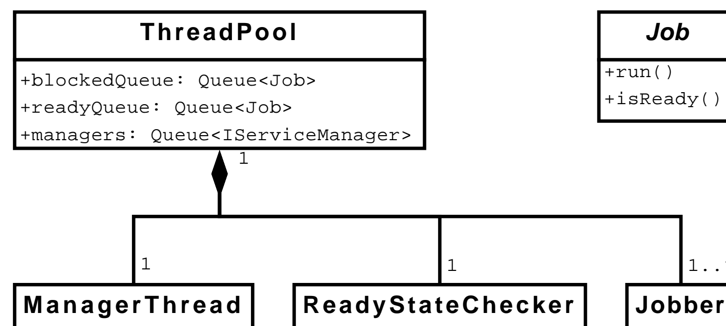

Abbildung 4.9: Die Struktur des Thread-Pools.

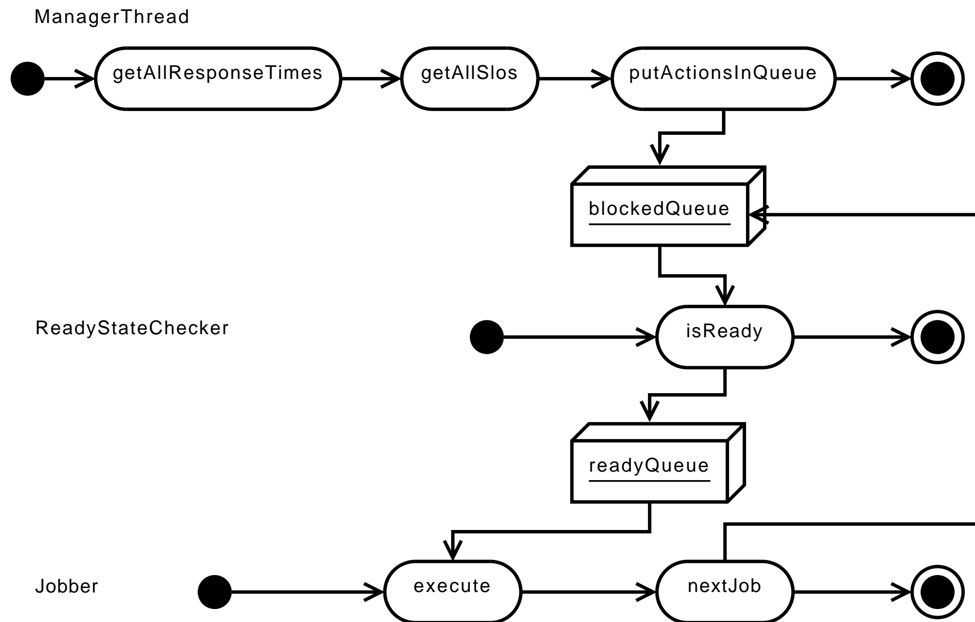

Abbildung 4.10: Das Aktivitätsdiagramm zum Thread-Pool.

Performance-Problemen in der Simulation möglich. Service-Manager und Strategie werden darum durch einen Thread-Pool ausgeführt (Abbildung 4.9). Der Thread-Pool besteht aus drei getrennten Thread-Arten.

Ein `ManagerThread` führt ausschließlich Service-Manager-Aktivitäten selbst aus. Die durch die Strategie auszuführenden Aktionen werden nicht länger von seinem separaten Thread ausgeführt, sondern als Jobs in eine Warteschlange geschrieben (Abbildung 4.10).

Da nicht alle Jobs sofort ausführbar sind, wird jeder dort enthaltene Job wiederkehrend von einem separaten `ReadyStateChecker`-Thread überprüft, ob er nun ausführbar ist. Ausführbare Jobs werden anschließend in eine weitere Warteschlange verschoben und dort von einem `Jobber` ausgeführt. Ein Job kann weitere nachfolgende Jobs in die `blockedQueue`-Warteschlange schreiben.

4.2.3 Strategie

Die hier beschriebene Strategie implementiert das in Abschnitt 3.5 entwickelte SLO-Spiel, inklusive beiden Verallgemeinerungen aus Abschnitt 3.5.4. Prinzipiell implementiert es also den in Abschnitt 3.5.1 definierten Ablauf des Spiels. Der Ablauf muss in der Realität erweitert werden, da im ursprünglichen Ablauf die vergangene Zeit keine Rolle spielt:

1. Bestimmung der Antwortzeit $t_{response}(a)$ und Vergleich mit SLO $slo(a)$.
2. Ein Agent i wird Verkäufer, falls $t_{response}(a) < slo(a)$, andernfalls Käufer.
3. Jeder Verkäufer ermittelt seinen Reservierungspreis, die zu versteigerndn SLO-Anteile und die *Ablaufzeit der Auktion*. Anschließend ist der Verkäufer bereit seine Auktion den Käufern bekannt zu machen. Dabei wird der Reservierungspreis, die Anzahl, die zu vergebender SLO-Anteile und die Ablaufzeit der Auktion mitgeteilt.
4. *Jeder Käufer sendet an alle Agenten eine Suchanfrage nach Auktionen und wartet eine bestimmte Zeitspanne auf Antworten von Verkäufern.*
5. Jeder Käufer gibt ein Gebot an einen Verkäufer ab.
6. Jeder Verkäufer ermittelt nach Beendigung der Auktion das Ergebnis und teilt es den jeweiligen Käufern mit.

Zustandsautomat

Jeder Agent durchläuft während dem Ablauf der Management-Aktion verschiedene Zustände. Zuerst wird wie in Abbildung 4.11 dargestellt ein Agent entweder Käufer oder Verkäufer.

Ein Verkäufer folgt anschließend dem in Abbildung 4.12 dargestellten Zustandsmodell ein Käufer dem in 4.13.

Der Verkäufer fängt sofort an, eine Auktion zu starten, d.h. er ermittelt Reservierungspreis, zu verkaufende SLO-Anteile und Ablaufzeit. Sobald er die Parameter der Auktion ermittelt hat, geht er in den nächsten Zustand über, in welchem der Verkäufer auf

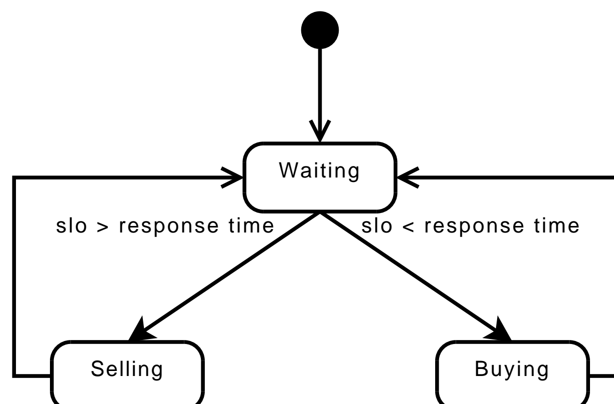

Abbildung 4.11: Die übergeordneten Zustände eines Agenten.

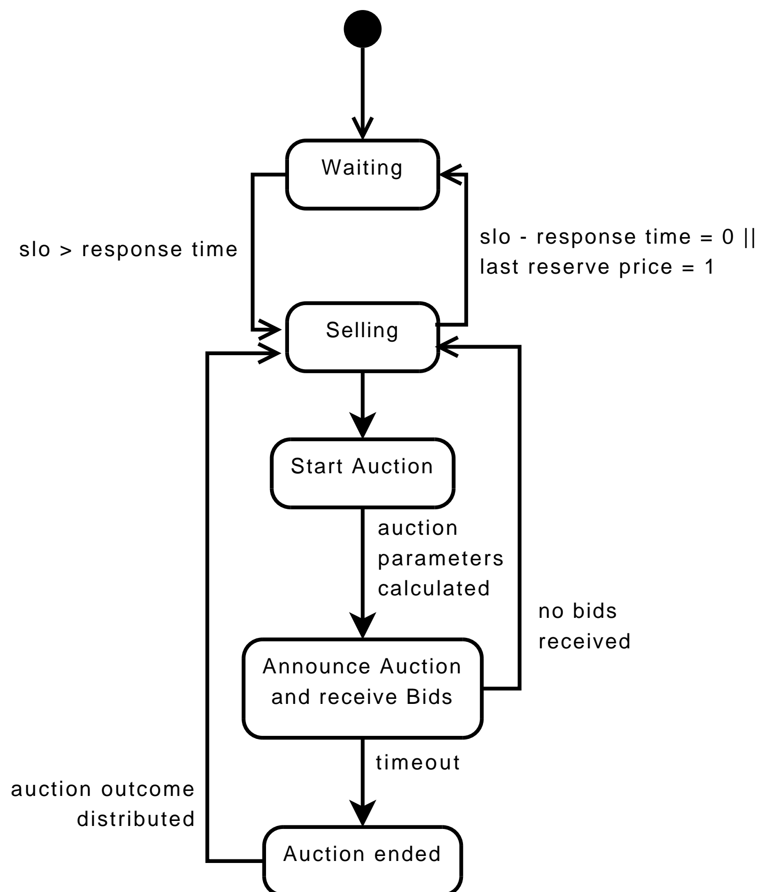

Abbildung 4.12: Das Zustandsmodell eines Verkäufers.

Nachrichten von Käufern wartet. Es gibt zwei Nachrichten die ein Käufer dem Verkäufer schicken kann:

- Anfrage nach Auktionen
- Gebot

Auf eine Anfrage nach Auktionen sendet der Verkäufer die Auktionsparameter an den anfragenden Käufer. Wird ein Gebot empfangen wird dies gespeichert.

Ein Verkäufer bleibt solange in diesem Zustand, bis die Ablaufzeit der Auktion verstrichen ist. Ist die Auktion beendet, wird das Ergebnis der Auktion ermittelt und an alle Käufer das Ergebnis verteilt. Das versendete Ergebnis besteht immer in der Anzahl der zugeteilten SLO-Anteile und im zu zahlenden Preis.

Sind alle Ergebnisse verteilt, kehrt der Verkäufer in den Ausgangszustand als Verkäufer zurück. Wurden nicht alle SLO-Anteile verkauft, wird der Verkäufer eine neue Auktion

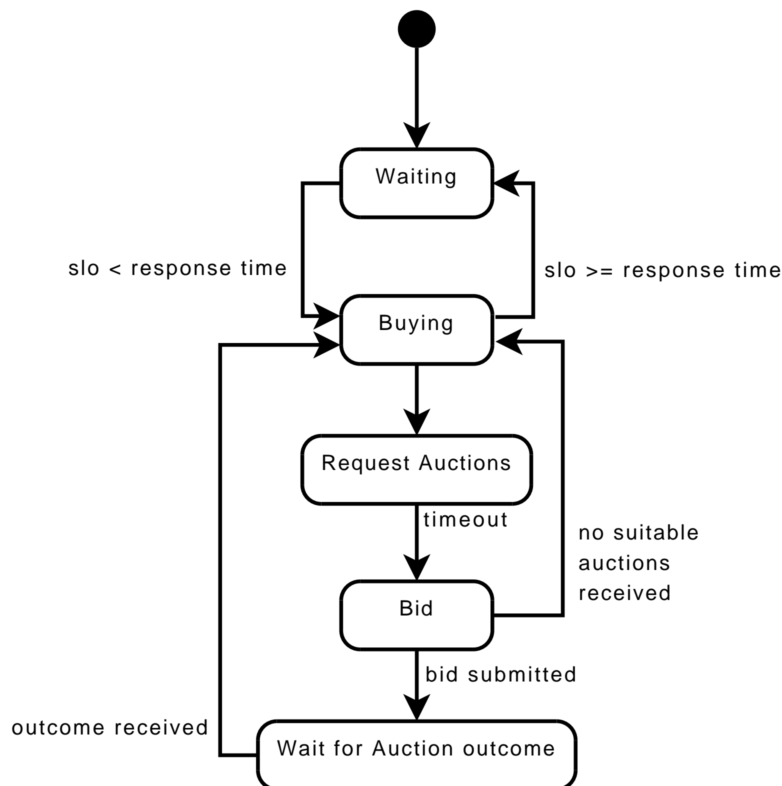

Abbildung 4.13: Das Zusstandsmodell eines Käufers.

erstellen. Wurden alle SLO-Anteile verkauft, kehrt der Verkäufer in einen Wartezustand zurück, bis der Service-Manager eine weitere Management-Aktion anstößt.

Der Käufer fängt damit an allen ihm bekannten Agenten eine Anfrage nach Auktionen zu versenden. Anschließend wartet er eine definierte Zeitspanne bis er versucht auf eine Auktion zu bieten.

Wurden keine passenden Auktionen übermittelt, weil beispielsweise bei allen Auktionen der Reservierungspreis zu hoch war oder die Auktion bereits abgelaufen war, als die Nachricht verarbeitet wurde, kehrt der Käufer in seinen Ausgangszustand zurück und fängt von vorne an. Wurde dagegen mindestens eine passende Auktion gefunden, wählt der Käufer davon eine Auktion zufällig aus, ermittelt sein Gebot und sendet das Gebot an den Verkäufer.

Anschließend wartet der Käufer auf ein Ergebnis der Auktion. Dies enthält entweder die zugeteilten SLO-Anteile und den zu zahlenden Preis, oder das Gebot kam zu spät beim Verkäufer an (z.B. wegen zu großer Nachrichtenlaufzeit) und der Käufer wird deshalb benachrichtigt.

Der Käufer kehrt nachdem er das Ergebnis erhalten hat zum Ausgangszustand zurück und

überprüft, ob er genügend SLO-Anteile erworben hat. Ist dies der Fall kehrt der Käufer zum Wartezustand zurück, bis der Service-Manager eine weitere Management-Aktion anstößt. Wurden nicht genügend SLO-Anteile erworben, wird das Zustandsmodell erneut durchlaufen.

Während sich ein Agent in einem bestimmten Zustand befindet, führt er nur eine gleichförmige Aktivität durch, die durch ein `Job`-Objekt repräsentiert wird, welches durch den Thread-Pool mit Hilfe eines `Jobbers` ausgeführt wird. Beim Zustandsübergang erstellt eine Aktivität ein weiteres `Job`-Objekt, außer für den Fall einer Rückkehr zum Wartezustand..

Gruppenkommunikation

Für die Kommunikation mit anderen Strategien wird ein Gruppenkommunikationsmechanismus benutzt (vgl. Abschnitt 2.4 und Abschnitt 3.5.3). Für jede Aktivität tritt eine Strategie einer Gruppe für die Sequenz oder Parallelausführung in der die Aktivität enthalten ist bei. Ist die Gruppe zu der man beigetreten ist eine Sequenz können alle Gruppenmitglieder untereinander SLO-Anteile auf Basis des SLO-Spiels austauschen.

Das zugrunde liegende Gruppenkommunikations-System wird so konfiguriert, dass folgende Eigenschaften gelten (vgl. Abschnitt 2.4):

- Mitglieder-Service:
  - Sichten werden nur installiert, wenn der Agent selbst in der Sicht enthalten ist.
  - Es werden nur neuere Sichten installiert.
  - Es muss eine Sicht vor dem Empfang von Nachrichten installiert sein.
  - Durch eine Partitionierung können alle entstandenen Partitionen separat weiterarbeiten. Partitionen können separat weiterarbeiten, können dann allerdings nur SLO-Anteile zwischen den in der Partition verbliebenen Agenten mittels Auktion austauschen. Das SLO-Spiel erfragt für jede Management-Aktion den Zustand aller beteiligten Agenten erneut, deshalb ist die Vereinigung von Partitionen trivial, sie passiert automatisch bei der nächsten Management-Aktion.
- Nachrichtenauslieferung:

  - An die Nachrichtenauslieferung werden keine gesonderten Anforderungen gestellt, d.h. ob eine Nachricht in einer bestimmten Sicht erhalten wird oder nicht, ist nicht interessant. Alle Agenten an die durch eine Multicast-Kommunikation eine Anfrage gestellt wird, besitzen einen unabhängigen Zustand. Deshalb ist es nicht wichtig, ob alle Agenten in einer Sicht die gleichen Nachrichten erhalten.

- Nachrichtenreihenfolge:
  - Es ist keine spezielle Nachrichtenreihenfolge nötig, denn Nachrichten sind entweder unabhängig von allen anderen empfangenen Nachrichten oder nur von einer bereits empfangenen Nachricht abhängig.
  - Da das eingesetzte Gruppenkommunikations-System grundsätzlich eine FIFO-Ordnung herstellt, wird diese genutzt.

- Lebendigkeit:
  - Es werden alle in Abschnitt 2.4.5 definierten Eigenschaften eingehalten.

Vertreter in zusammengesetzten Aktivitäten

Aus jeder Sequenz oder Parallelausführung wird außerdem ein Vertreter gewählt, der die zusammengesetzte Aktivität in einer darüberliegenden weiteren Sequenz oder Parallelausführung vertritt.

Der Vertreter erzeugt einen virtuellen Service und SLO-Container (Abbildung 4.14). Dieser virtuelle Service kann von einem Service-Manager wie ein normaler Service genutzt

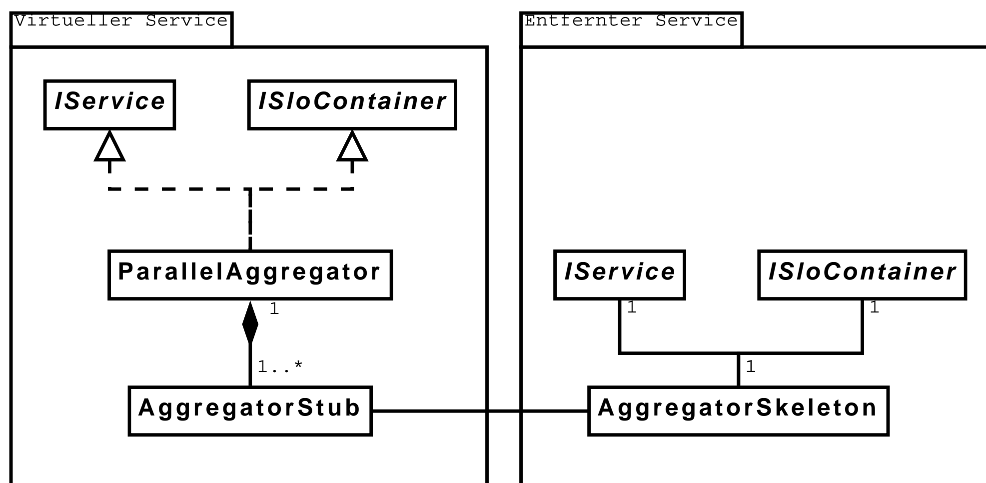

Abbildung 4.14: Der Aufbau des `ParallelAggregators`.

werden. Der Vertreter selbst greift auf einen Stub zurück, der über den Gruppenkommunikationsmechanismus mit den tatsächlichen Services mittels eines Skeleton kommuniziert.

Erweiterung für lokale Auktionen

Um die in Abschnitt 3.5.4 beschriebene Erweiterung zur lokalen Umverteilung von Antwortzeitanteilen auf einem Service zu unterstützen, muss die bisherige Strategie leicht erweitert werden.

Auktionen die von einem Agenten durchgeführt werden, der mit anderen Agenten gemeinsam auf einem Service ausgeführt wird, können vereinfacht über eine Shared-Memory-Kommunikation bekannt gemacht werden. Dadurch erübrigt sich der Prozess des Bekanntmachens einer Auktion durch aufwendige Kommunikationsschritte.

Die zweite Änderung betrifft die Berechnung von benötigten und tatsächlich zugeteilten Anteilen, da diese sich nach dem in Abschnitt 3.4.2 beschriebenen Modell verhalten. Es lassen sich versteigerte SLO-Anteile anhand dieses Modells in Antwortzeitanteile umrechnen.

Da ein Gebot immer für die benötigten SLO-Anteile berechnet wird, muss zu erst ermittelt werden wie groß der Antwortzeitanteil ist. Anschließend muss dies in SLO-Anteile für den Verkäufer umgerechnet werden, damit das Gebot zusammen mit anderen Geboten sinnvoll bei der Auktion mit beachtet werden kann. Wurden dem Käufer durch die Auktion Anteile zugeteilt, erfolgt der umgekehrte Weg.

4.2.4 Simulations-Controller

Der Simulations-Controller ist für die Konfiguration, Steuerung und Überwachung der Simulation zuständig. Als solches ist er für die Erzeugung von Services und dazugehörigen Service-Managern als auch für die Änderung von Parametern zuständig.

Zur Erzeugung von Services wird die Simulations-Umgebung benachrichtigt (Abbildung 4.15). Die Simulations-Umgebung erzeugt daraufhin zuerst einen neuen simulierten Service, anschließend einen Service-Manager. Der Service-Manager erzeugt seinerseits eine Strategie, welche er in Zukunft für Management-Aktionen nutzen wird.

Anschließend wird der Management-Thread gestartet. Ohne den Thread-Pool würde hier ein eigener Thread erzeugt, der für die Ausführung des Service-Managers zuständig ist. Mit dem Thread-Pool wird der Service-Manager in der Queue des Thread-Pools abgelegt.

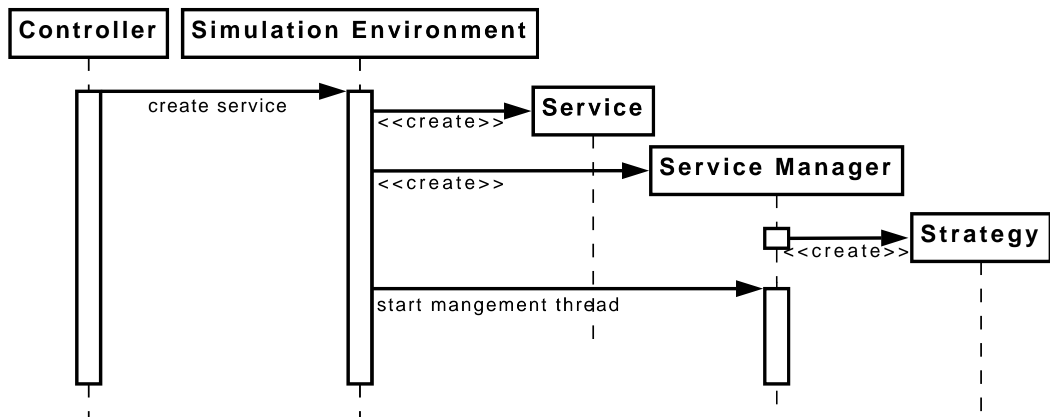

Abbildung 4.15: Die Erzeugung eines Services.

Der Simulations-Controller führt Änderungen durch, indem er direkt die Services oder Service-Manager anspricht (Abbildung 4.15). Diese setzen die Änderungen eigenständig um. Auf diese Weise werden beispielsweise die vorhandenen Aktivitäten den Service-Managern zugeteilt, oder die Antwortzeit eines Services manipuliert.

Die GUI des Simulations-Controller wird als Swing-Applikation vorgesehen. Für die Verarbeitung von Ereignissen wird deshalb das dort übliche Observer-Pattern genutzt, d.h. implementiert die `ActionListener`-Schnittstelle für jede angebotene Aktion.

Weiterhin müssen Elemente der GUI aktualisiert werden. Die Informationsbeschaffung durch den Simulations-Controller erfolgt grundsätzlich mittels Abfragen vom Simulations-Controller an Teile der Simulations-Umgebung. Für kontinuierliche Änderungen, wie die Übersichtsfunktionen, werden periodisch aus der Simulationsumgebung Informationen abgerufen und die Anzeigen damit aufgefrischt. Für statische Informationen besitzt der Simulations-Controller ein `ConfigData`-Singleton, das als Informationsquelle für

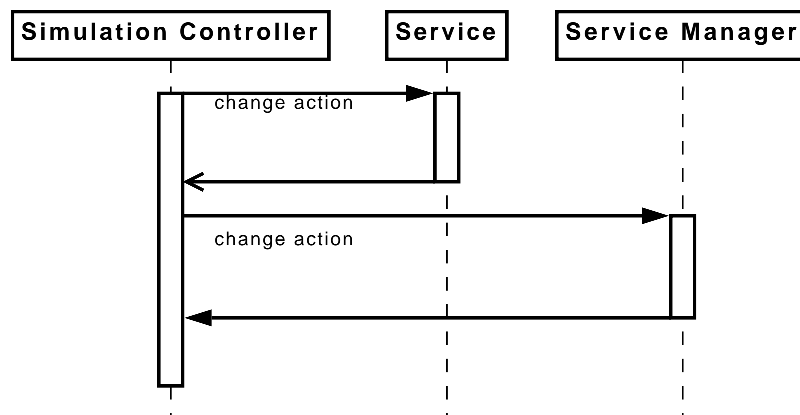

Abbildung 4.16: Veränderungen an der Simulation.

den Simulations-Controller dient und bei Änderungen durch den Simulations-Controller an der Simulationsumgebung aktualisiert wird.

4.2.5 Verteiltes System

Für den Entwurf des SLO-Spiels als verteiltes System wird im Wesentlichen die Funktionalität des Simulations-Controllers auf die einzelnen teilnehmenden alleinstehenden Services aufgeteilt.

Um identische Konfigurationen für die Simulation als auch für das verteilte System nutzen zu können wird die gleiche Konfigurationen an alle teilnehmenden Services verteilt. Zusätzlich wird jeder Teilnehmer darüber informiert, welchen Service aus der Konfiguration er übernimmt und welche Aktivitäten der Workflows er zugewiesen bekommt. Aus der vollständigen Konfiguration extrahiert sich jeder Service die für in relevanten Daten und verwirft die restlichen Konfigurationsdaten.

4.3 Informationsmodell

Um die Simulation effektiv steuern und Ergebnisse auswerten zu können, wird ein Informationsmodell festgelegt, das den Austausch von Informationen mit der Simulations-Umgebung ermöglicht. Dieses Informationsmodell wird in zwei grundsätzliche Bereiche geteilt, die Konfiguration und Steuerung, sowie Auswertung.

4.3.1 Konfigurations- und Steuerungsinformationen

Für die Konfiguration und Steuerung der Simulation sind Workflows auf Basis des in Abschnitt 3.2 definierten Modells zu definieren:

- atomare Aktivitäten
- Parallelausführungen
- Sequenzen
- Schleifen

Jeder atomaren Aktivität wird außerdem ein Service über eine Service-ID zugeordnet. Jedem Workflow wird weiterhin ein SLA zugeordnet, das für das SLO-Spiel nur aus:

- einer maximalen Antwortzeit und
- einer Priorität des Workflows

besteht. Aus der Priorität wird durch den Simulations-Controller bzw. den alleinstehenden Service ein virtueller Geldbetrag ermittelt der jeder Aktivität zusteht.

Es wird weiterhin eine Konfiguration der simulierten Services vorgenommen, diese enthält:

- Anzahl der Services
- Eine Service-ID für jeden Service
- Ein Antwortzeitmodell für jeden simulierten Service

4.3.2 Trace-Log

Das Trace-Log wird durch einen Testlauf erzeugt und enthält die in Abschnitt 3.6 definierten Messgrößen:

- Start- und Endzeit eines Bieters oder Auktionators
- Start- und Endzeit einer Auktion
- Reservierungspreis einer Auktion und durch eine Auktion zu verkaufende SLO-Anteile
- Zeitpunkt der Abgabe des Gebots, tatsächlicher Gebotspreis und gewünschte SLO-Anteile
- Zeitpunkt einer Nachrichtenübermittlung

Das Trace-Log wird durch die Strategien erzeugt, welche alle Informationen zur Ermittlung dieser Messgrößen besitzen. Für den Fall der Simulations-Umgebung, werden alle Ereignisse in das gleiche Trace-Log geschrieben. Im verteilten Fall wird für jeden Service ein Trace-Log erstellt. Zur Auswertung müssen diese verteilten Trace-Logs zusammengefasst werden.

Kapitel 5

Implementierung

Im Weiteren sollen Aspekte der Implementierung dargestellt werden, die den Entwurf aus dem vorangegangenen Kapitel umsetzen und weiter verfeinern.

Es wird darauf eingegangen wie die Simulation und das verteilte System konfiguriert wird und wie das Trace-Log erstellt wird. Außerdem werden die genutzten Kommunikationssysteme- und Protokolle und deren Anwendung beschrieben.

Ein detaillierter Blick in die Funktionalitäten der grafischen Benutzeroberfläche rundet die Darstellung der Implementierung ab.

Eine kurze Codestatistik gibt zusätzlich einen Überblick über den Implementierungsaufwand der Simulation und der verteilten Umgebung.

5.1 Konfiguration

Die Konfiguration der Simulation erfolgt in zwei Teilen:

- Service-Definition
- Workflow-Definitionen

Für den verteilten Fall gibt es zwei weitere Teil-Konfigurationen:

- Konfiguration des einzelnen Service
- Konfiguration der Gruppenkommunikation

Das Schema aller Konfigurations-Teile ist per XML-Schema [TBMM04, BM04] festgelegt. Das XML-Schema für Service-Definitionen:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4              targetNamespace="http://xml.netbeans.org/schema/serviceModel"
5              xmlns:tns="http://xml.netbeans.org/schema/serviceModel"
6              elementFormDefault="qualified">
7      <xsd:complexType name="service">
8          <xsd:attribute name="serviceID" type="xsd:int" use="required"/>
9          <xsd:attribute name="responseTime" type="xsd:long" use="required"/>
10     </xsd:complexType>
11     <xsd:complexType name="responseTimeChange">
12         <xsd:attribute name="startTime" type="xsd:long"/>
13         <xsd:attribute name="newResponseTime" type="xsd:long"/>
14         <xsd:attribute name="random" type="xsd:boolean"/>
15         <xsd:attribute name="lowerBoundStartTime" type="xsd:long"/>
16         <xsd:attribute name="upperBoundStartTime" type="xsd:long"/>
17         <xsd:attribute name="lowerBoundResponseTime" type="xsd:long"/>
18         <xsd:attribute name="upperBoundResponseTime" type="xsd:long"/>
19     </xsd:complexType>
20     <xsd:complexType name="responseTimeModel">
21         <xsd:sequence>
22             <xsd:element name="responseTimeChange" type="tns:responseTimeChange"
                   maxOccurs="unbounded"></xsd:element>
23         </xsd:sequence>
24         <xsd:attribute name="serviceID" type="xsd:int" use="required"/>
25     </xsd:complexType>
26     <xsd:element name="serviceModel">
27         <xsd:complexType>
28             <xsd:sequence maxOccurs="unbounded">
29                 <xsd:element name="service" type="tns:service"
                       minOccurs="1"></xsd:element>
30                 <xsd:element name="responseTimeModel" type="tns:responseTimeModel"
                       minOccurs="0"></xsd:element>
31             </xsd:sequence>
32         </xsd:complexType>
33     </xsd:element>
34 </xsd:schema>
```

Listing 5.1: Das XML-Schema für Service-Definitionen

Das Wurzel-Element jedes XML-Dokuments, welches eine Service-Definition darstellt ist das `serviceModel`-Element. Dieses enthält mindestens einen `service`-Element, welches einen Service beschreibt. Das Service-Element besitzt zwei Attribute die Service-ID (`serviceID`) und eine initiale Antwortzeit (`responseTime`).

Es kann außerdem für jeden Service ein Antwortzeitmodell durch die Nutzung eines `responseTimeModel`-Element erstellt werden. Ein Antwortzeitmodell enthält Einträ-

ge, welche die Änderung der Antwortzeit zu bestimmten Zeitpunkten auf feste Antwortzeiten festlegt, oder es wird aus einem festzulegenden Intervall zufällig eine Startzeit für eine neue Antwortzeit gezogen. Die neue Antwortzeit wird dann ebenfalls zufällig aus einem zweiten Intervall gezogen.

Im Weiteren wird das Schema für Workflow-Definitionen gezeigt:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4              targetNamespace="http://xml.netbeans.org/schema/workflow"
5              xmlns:tns="http://xml.netbeans.org/schema/workflow"
6              elementFormDefault="qualified">
7      <xsd:complexType name="workflow">
8          <xsd:choice>
9              <xsd:element name="sequence" type="tns:sequence"></xsd:element>
10             <xsd:element name="parallel" type="tns:parallel"></xsd:element>
11             <xsd:element name="activity" type="tns:activity"></xsd:element>
12         </xsd:choice>
13         <xsd:attribute name="maxResponseTime" type="xsd:long" use="required"/>
14         <xsd:attribute name="priority" type="xsd:int" use="required"/>
15         <xsd:attribute name="name" type="xsd:string" use="required"/>
16     </xsd:complexType>
17     <xsd:complexType name="activity">
18         <xsd:attribute name="assignedService" type="xsd:int" use="required"/>
19     </xsd:complexType>
20     <xsd:complexType name="sequence">
21         <xsd:sequence minOccurs="1" maxOccurs="unbounded">
22             <xsd:element name="parallel" type="tns:parallel" maxOccurs="1"
                   minOccurs="0"></xsd:element>
23             <xsd:element name="loop" type="tns:loop" maxOccurs="1"
                   minOccurs="0"></xsd:element>
24             <xsd:element name="activity" type="tns:activity" minOccurs="0"
                   maxOccurs="1"></xsd:element>
25         </xsd:sequence>
26     </xsd:complexType>
27     <xsd:complexType name="parallel">
28         <xsd:sequence minOccurs="1" maxOccurs="unbounded">
29             <xsd:element name="sequence" type="tns:sequence"
                   minOccurs="0"></xsd:element>
30             <xsd:element name="loop" type="tns:loop" minOccurs="0"></xsd:element>
31             <xsd:element name="activity" type="tns:activity"
                   minOccurs="0"></xsd:element>
32         </xsd:sequence>
33     </xsd:complexType>
34     <xsd:element name="workflow" type="tns:workflow"/>
35     <xsd:complexType name="loop">
36         <xsd:choice>
37             <xsd:element name="sequence" type="tns:sequence"></xsd:element>
38             <xsd:element name="parallel" type="tns:parallel"></xsd:element>
39             <xsd:element name="activity" type="tns:activity"></xsd:element>
40         </xsd:choice>
```

```
41     </xsd:complexType>
42 </xsd:schema>
```

Listing 5.2: Das XML-Schema für Workflow-Definitionen

Das Schema orientiert sich an der Definition eines Workflows aus Abschnitt 3.2. Jeder Workflow enthält mindestens eine Aktivität. Ist diese Aktivität eine zusammengesetzte Aktivität, müssen mindestens eine weitere Aktivitäten enthalten, generell gilt dies immer für eine zusammengesetzte Aktivität. Im Gegensatz dazu können atomare Aktivitäten keine weiteren Aktivitäten enthalten.

Eine weitere Einschränkung zur Definition aus Abschnitt 3.2 ist, dass eine zusammengesetzte Aktivität keine weitere zusammengesetzte Aktivitäten des gleichen Typs enthalten kann. Dies schränkt die Ausdrucksfähigkeit nicht ein, da geschachtelte Aktivitäten des gleichen Typs immer zu einer einzelnen zusammengesetzten Aktivität zusammengefasst werden können.

Für einzelne Service im verteilten Modell muss ebenfalls eine Konfiguration erfolgen. Eine Konfigurationdatei nach dem hier dargestellten Schema teilt dem Service seine Service-ID, die Lage weiterer Konfigurationsdateien, die verwendete Strategie und für jeden bekannten Workflow die ID einer Aktivität mit, welche diesem Service zugeteilt wird:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4              targetNamespace="http://xml.netbeans.org/schema/nodeconfig"
5              xmlns:tns="http://xml.netbeans.org/schema/nodeconfig"
6              elementFormDefault="qualified">
7
8      <xsd:element name="nodeConfig">
9          <xsd:complexType>
10             <xsd:sequence>
11     <xsd:element name="serviceModel">
12         <xsd:complexType>
13             <xsd:attribute name="strategyName" type="xsd:string" use="required"/>
14             <xsd:attribute name="configFile" type="xsd:string" use="required"/>
15             <xsd:attribute name="serviceID" type="xsd:int" use="required"/>
16         </xsd:complexType>
17     </xsd:element>
18     <xsd:element name="tcpConfiguration">
19         <xsd:complexType>
20             <xsd:attribute name="configFile" type="xsd:string" use="required"/>
21         </xsd:complexType>
22     </xsd:element>
23     <xsd:element name="workflowModel" maxOccurs="unbounded">
24         <xsd:complexType>
25             <xsd:attribute name="activityID" type="xsd:string" use="required"/>
26             <xsd:attribute name="configFile" type="xsd:string" use="required"/>
```

```
27          </xsd:complexType>
28      </xsd:element>
29      </xsd:sequence>
30          </xsd:complexType>
31      </xsd:element>
32  </xsd:schema>
```

Listing 5.3: Das XML-Schema für Service-Knoten der verteilten Implementierung

Um im verteilten Fall mittels des Gruppenkommunikations-Systems potenzielle Kommunikationsteilnehmer zu finden, müssen die bekannten Teilnehmer dem Service mitgeteilt werden. Konfiguriert werden muss dazu der Host und der TCP-Port, sowie der Gruppenname, welcher einer `containerID` entspricht. Die Konfiguration der eigenen Gruppenkommunikation erfolgt über die Aktivitäts-ID. Die Aktivitäts-ID wurde dem Service durch die übergeordnete Konfigurations-Datei für einen Service-Knoten mitgeteilt.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4              targetNamespace="http://xml.netbeans.org/schema/tcpConfiguration"
5              xmlns:tns="http://xml.netbeans.org/schema/tcpConfiguration"
6              elementFormDefault="qualified">
7      <xsd:element name="tcpConfiguration">
8          <xsd:complexType>
9              <xsd:sequence>
10                 <xsd:element name="container" maxOccurs="unbounded">
11                     <xsd:complexType>
12                         <xsd:sequence>
13                             <xsd:element name="node" maxOccurs="unbounded">
14                                 <xsd:complexType>
15                                     <xsd:attribute name="host" type="xsd:string"
                                           use="required"/>
16                                     <xsd:attribute name="port" type="xsd:unsignedShort"
                                           use="required"/>
17                                     <xsd:attribute name="activityID" type="xsd:string"
                                           use="required"/>
18                                 </xsd:complexType>
19                             </xsd:element>
20                         </xsd:sequence>
21                         <xsd:attribute name="containerID" type="xsd:string"
                               use="required"/>
22                     </xsd:complexType>
23                 </xsd:element>
24             </xsd:sequence>
25         </xsd:complexType>
26     </xsd:element>
27 </xsd:schema>
```

Die Konfiguration wird sowohl vom Simulations-Controller als auch von einem alleinstehenden Service mit Hilfe des Apache Xerces2-J XML-Parsers [Apa08] geladen.

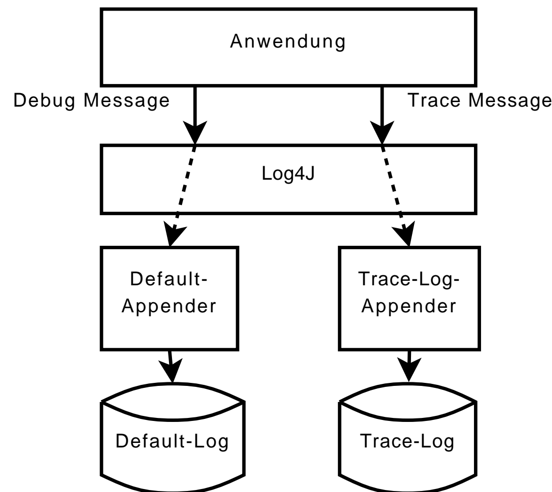

Abbildung 5.1: Gruppenbildung mehrerer Strategien.

In der Simulation werden anschließend durch den Simulations-Controller die Services in der Simulations-Umgebung erzeugt, bzw. die Aktivitäten an die Service-Manager zugewiesen. Bei den alleinstehenden Services wird dies durch jeden einzelnen Service jeweils selbst übernommen.

5.2 Auswertung

Die Auswertung besteht aus zwei Teilen:

- Erstellung eines Trace-Logs
- Konvertierung der Messwerte in Kenngrößen (vgl. Abschnitt 3.6)

Für die Erstellung des Trace-Logs wurde die Strategie mit Log4J [Gül02] instrumentiert [KS93]. Mit Log4J kann dazu ein separater sogenannte Logger erstellt werden, welcher über eine Konfigurationsdatei angewiesen wird mittels eines sogenannten Log-Appenders Ereignisse in ein separates Logfile zu schreiben (Abbildung 5.1).

Listing 5.4 zeigt einen Ausschnitt aus einem zu erstellten Trace-Log. Jeder Eintrag enthält einen absoluten Zeitstempel in Millisekunden, die ID der Aktivität und eine Beschreibung des Ereignisses.

```
1 1221931703554 /w0/s0/a12 bidder started, buy slo shares: 3
2 1221931703575 /w0/s0/a13 bidder started, buy slo shares: 3
3 1221931703596 /w0/s0/a14 auctioneer started, sell slo shares: 7
```

```
4 1221931703780 /w0/s0/a13 Submit bid: 11 slo share: 3 auctionID:
      05f1e248-06bc-4e71-8e80-a4e12be94580
5 1221931703829 /w0/s0/a14 start calculate auction outcome, auction
      05f1e248-06bc-4e71-8e80-a4e12be94580
6 1221931703829 /w0/s0/a14 stop calculate auction outcome, auction
      05f1e248-06bc-4e71-8e80-a4e12be94580 income: 33 sold slo shares: 3
7 1221931703829 /w0/s0/a14 start distribute auction outcome, auction
      05f1e248-06bc-4e71-8e80-a4e12be94580
```

Listing 5.4: Ausschnitt aus einem Trace-Log

Die Konvertierung des Trace-Logs in Kenngrößen erfolgt durch einen Konverter, welcher die Daten aufbereitet. Die Anzahl aktiver Bieter pro Zeiteinheit wird ermittelt, indem gezählt wird:

- wieviele Bieter in dieser Zeiteinheit gestartet sind und
- wieviele Bieter sich in dieser Zeiteinheit beendet haben.

Die Zahl der gestarteten Bieter wird zu der Anzahl aktiver Bieter der vorherigen Zeiteinheit addiert, die Anzahl der beendeten Bieter wird subtrahiert. Solange die Genauigkeit dieser Messung höher ist, als die Zeitdauer der Aktivität eines Bieters minimal ist, werden hier alle Bieter erfasst.

Für Verkäufer und Auktionen wird ebenso vorgegangen. Für die Anzahl Nachrichten pro Zeiteinheit, wird in jeder Zeiteinheit die Summe aller Nachrichten gebildet. Da Netzwerklasten in der Regel in Nachrichten pro Sekunde gemessen werden, wird als Zeiteinheit für die Ermittlung der Nachrichten immer 1 Sekunde angesetzt.

5.3 JMX

Zur Kommunikation zwischen dem Simulations-Controller und der Simulations-Umgebung werden die Java Management Extensions (JMX) [FL01] eingesetzt. JMX ist ein sehr mächtiges Framework zum Management von Anwendungen. Die durch den Simulations-Controller genutzten Features werden hier kurz vorgestellt.

Über JMX kann auf entfernte Objekte zugegriffen werden. Ein sogenanntes Standard-MBean folgt der Spezifikation von Java Beans [EK02]. Ein Standard-MBean besitzt

- Attribute und
- Operationen.

Damit eine Klasse als Standard-MBean erkannt werden kann, müssen verschiedene Voraussetzungen erfüllt sein:

- Eine Klasse `*` implementiert eine Schnittstelle `*MBean`.
- Für ein Attribut `*` definiert die MBean-Schnittstelle `get*()`- oder `set*()`-Methoden. Ein Attribut das nur eine `get*()`-Methode besitzt ist ein schreibgeschütztes Attribut. Ein Attribut das nur eine `set*()`-Methode besitzt, kann nur geschrieben werden. Sind beide Methoden vorhanden, kann das Attribut gelesen und geändert werden.
- Operationen sind beliebige andere Methoden.
- Parameter und Rückgabewerte aller Methoden müssen die `Serializable`-Schnittstelle implementieren.

Für die Simulationsumgebung werden insgesamt vier MBean-Schnittstellen definiert:

- `SimulationEnvironmentMBean`
- `SimulationServiceMBean`
- `ServiceManagerMBean`
- `VickreyAuctionStrategyMBean`

Die `SimulationEnvironmentMBean`-Schnittstelle definiert im Wesentlichen eine Schnittstelle mit deren Hilfe Services angelegt werden können, die globale von allen Service-Managern genutzte Strategie festlegt werden kann. Außerdem kann darüber die Simulation gestartet und beendet werden.

Über `SimulationServiceMBean`-Schnittstelle lässt sich die Antwortzeit des simulierten Services abfragen oder manipulieren, sowie Antwortzeitanteile für einzelne Aktivitäten ändern.

Dem Service-Manager kann über die `ServiceManagerMBean`-Schnittstelle eine neue Aktivität zugewiesen werden, oder es kann eine vorhandene Aktivität gelöscht werden. Es kann weiterhin der Status einer Aktivität abgefragt werden, entweder die SLO-Einhaltung, oder das SLO selbst.

Mittels der `VickreyAuctionStrategyMBean`-Schnittstelle weist der Simulations-Controller einzelnen Agenten Geld zu.

Um auf ein Objekt mittels der MBean-Schnittstelle über JMX zugreifen zu können muss das Objekt bei einem `MBeanServer` registriert werden. Auf dieses Objekt kann anschließend entweder lokal über den gleichen `MBeanServer` oder entfernt über eine `MBeanServerConnection` zugegriffen werden. Für den entfernten Zugriff erzeugt JMX automatisch eine `Proxy`-Klasse mittels des Reflection-Mechanismus von Java.

Auf Seiten des Simulations-Controllers wird mittels einer `JMXConnectorFactory` eine Verbindung zur Simulations-Umgebung hergestellt. Als Stellvertreter für diese Verbindung erhält man von der `JMXConnectorFactory` ein `MBeanServerConnection`-Objekt, über das der Simulations-Controller auf die Simulations-Umgebung zugreifen kann.

Der Aufbau und die Verwaltung der JMX-Verbindung vom Simulations-Controller zur Simulations-Umgebung wird von einem `JMXManager`-Singleton gekapselt.

5.4 Gruppenkommunikation und JGroups

Als Gruppenkommunikations-System wird JGroups eingesetzt (vgl. Abschnitt 2.4.6). Da JGroups allerdings einige Threads pro Gruppenmitglied erzeugt, steigt der Ressourcenbedarf schnell stark an, wodurch für eine größere Anzahl von Teilnehmern am SLO-Spiel in der Simulations-Umgebung eine Alternative implementiert wurde. Im verteilten Fall wird weiterhin JGroups eingesetzt.

Die im Entwurf (Abschnitt 4.2.3) beschrieben Eigenschaften werden mit folgender Konfiguration des JGroups-Protokoll-Stacks erreicht:

```
1  TCP(start_port=$channelPort$
2      loopback=true;
3      recv_buf_size=20000000;
4      send_buf_size=640000;
5      discard_incompatible_packets=true;
6      max_bundle_size=64000;
7      max_bundle_timeout=30;
8      use_incoming_packet_handler=true;
9      enable_bundling=true;
10     use_send_queues=false;
11     sock_conn_timeout=300;
12     skip_suspected_members=true;
13
14     use_concurrent_stack=true;
15
16     thread_pool.enabled=true;
17     thread_pool.min_threads=1;
```

```
18      thread_pool.max_threads=1;
19      thread_pool.keep_alive_time=5000;
20      thread_pool.queue_enabled=true;
21      thread_pool.queue_max_size=1000;
22      thread_pool.rejection_policy=Run;
23      oob_thread_pool.enabled=true;
24      oob_thread_pool.min_threads=1;
25      oob_thread_pool.max_threads=1;
26      oob_thread_pool.keep_alive_time=5000;
27      oob_thread_pool.queue_enabled=false;
28      oob_thread_pool.queue_max_size=100;
29      oob_thread_pool.rejection_policy=Run):
30  TCPPING(timeout=5000;
31          initial_hosts=$initialHosts$
32          port_range=1;
33          num_initial_members=10):
34  MERGE2(max_interval=10000;
35         min_interval=5000):
36  FD(timeout=10000;
37     max_tries=3):
38  VERIFY_SUSPECT(timeout=15000):
39  pbcast.NAKACK(max_xmit_size=60000;
40                use_mcast_xmit=false;
41                gc_lag=0;
42                discard_delivered_msgs=true;
43                retransmit_timeout=100,200,300,600,1200,2400,4800):
44  UNICAST(timeout=300,600,1200,2400,3600):
45  pbcast.STABLE(stability_delay=1000;
46                desired_avg_gossip=50000;
47                max_bytes=400000):
48  VIEW_SYNC(avg_send_interval=1000):
49  pbcast.GMS(print_local_addr=true;
50             join_timeout=3000;
51             join_retry_timeout=2000;
52             shun=true)
53  FRAG2(frag_size=60000)
```

Listing 5.5: Konfiguration von JGroups für TCP

Die einzelnen Protokolle entsprechen der Beschreibung des Beispiels in Abschnitt 2.4.6. Im Vergleich zum Beispiel in den Grundlagen, enthält dieser Protokoll-Stack weitere Protokolle und Optionen:

- Für TCP sind zusätzliche Optionen vorgesehen:
  - Es können Nachrichten in einem *Bundle* zusammengefasst werden, um bei sehr kleinen Nachrichten die Netzwerklast zu reduzieren.
  - Es wird ein Thread-Pool für diesen Channel verwendet. In der einfacheren Konfiguration werden für jedes Protokoll zwei Threads gestartet [Ban08b].

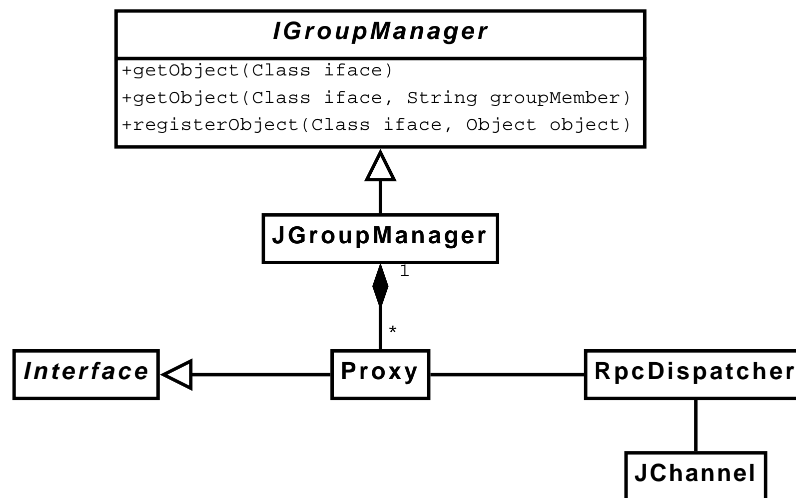

Abbildung 5.2: Struktur des Multicast-RPC-Diensts mit JGroups.

- Mit dem `UNICAST`-Protokoll wird die Nachrichtenreihenfolge von Unicast-Nachrichten sichergestellt. Auf Grund des Thread Pools, können sich Nachrichten auf ihrem Weg durch den Thread-Pool überholen. Dadurch ist die von TCP ansonsten gewährleistete Nachrichtenreihenfolge zerstört [Ban08b].
- Mit dem `VIEW_SYNC`-Protokoll wird erreicht, dass regelmäßig die Sicht aktualisiert wird. Dadurch wird ein Problem mit der zuverlässigen Installation von Sichten behoben [Ban08a].
- Das `FRAG2`-Protokoll fragmentiert große Nachrichten auf der Seite des Senders und setzt diese auf der Empfängerseite wieder zusammen.

Auf Basis von JGroups wird mit Hilfe des Building-Blocks `RpcDispatcher` und der `Proxy`-Klasse des Java-Reflection-Frameworks ein Multicast-RPC-Dienst implementiert (Abbildung 5.2).

Die `IGroupManager`-Schnittstelle stellt die folgenden Methoden zur Verfügung:

- `getObject()`
- `registerObject()`

Mit den beiden Methoden können Objekte für ein Unicast- oder Multicast-RPC erhalten oder registrieren werden. Die in der Arbeit implementierte JGroups-Implementierung `JGroupManager` der Schnittstelle `IGroupManager` implementiert diese Schnittstelle mittels eines `Proxy`-Objekts.

Ein `Proxy`-Objekt wird dynamisch mittels dem Java-Reflection-Framework angelegt und implementiert eine bestimmte Schnittstelle. Die gewünschte Schnittstelle wird in der `getObject()`-Methode angeben. Wird kein Gruppenmitglied angegeben wird ein Proxy-Objekt erzeugt, das über den `RpcDispatcher` ein Multicast-RPC durchführt. Wird ein Gruppenmitglied angegeben, wird über den `RpcDispatcher` ein Unicast-RPC durchgeführt.

Auf der Seite des Empfängers des RPCs muss ein Objekt registriert sein, dass die gewünschte Schnittstelle implementiert.

Für den JGroups-Ersatz wird die `IGroupManager`-Schnittstelle neu implementiert. Als Ersatz für den `RpcDispatcher` findet die Kommunikation über Methodenaufrufe zu den entsprechenden Objekten der anderen Gruppenmitgliedern statt. Dabei muss für den Multicast-RPC weiterhin ein Proxy-Objekt erstellt werden, welches alle Objekte mit der gewünschten Schnittstelle aller Gruppenmitglieder aufruft. Für den Unicast-RPC entfällt das Proxy-Objekt, da hier direkt auf das Objekt selbst zugegriffen werden kann.

5.5 Grafische Benutzeroberfläche

Die grafische Benutzeroberfläche ist in zwei Teile aufgeteilt:

- Konfiguration (Abbildung 5.3)
- Überwachung (Abbilding 5.4)

Die Konfiguration bietet die in Abschnitt 4.1 beschriebenen Konfigurations-Möglichkeiten. Wenn es sinnvoll ist, wird ein Dialog geöffnet, in dem eine Konfigurationsdatei ausgewählt werden kann. Die Konfiguration erfolgt auf jeden Fall in zwei Schritten. Zuerst müssen die Services angelegt werden, anschließend können Workflows hinzugefügt werden. Nachdem Services angelegt wurden, kann auch die Strategie verändert werden. Nach der Konfiguration folgt der Start des Simulationslaufs.

Änderungen der SLO-Einhaltung können über die Überwachungsoberfläche (Abbilding 5.4) beobachtet werden. In einem Workflow signalisiert die Farbe rot eine SLO-Verletzung, grün signalisiert eine SLO-Einhaltung (vgl. Abschnitt 4.1).

Soll der Simulationslauf beendet werden, kann dies wieder über die Konfigurationsoberfläche erfolgen. Nach erfolgtem Simulationslauf, müssen die Services und Workflows neu geladen werden. Bevor der Simulations-Controller geschlossen wird, sollte die Simulations-Umgebung heruntergefahren werden, da diese sonst am Leben bleibt.

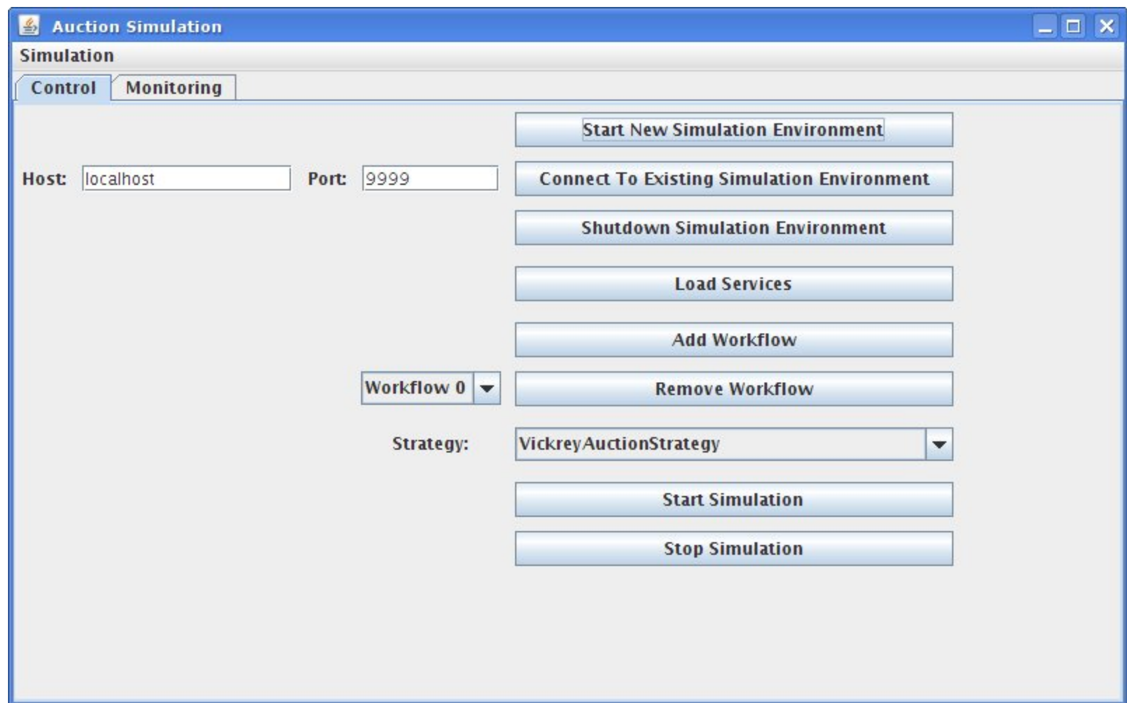

Abbildung 5.3: Die Konfigurationsoberfläche des Simulations-Controllers.

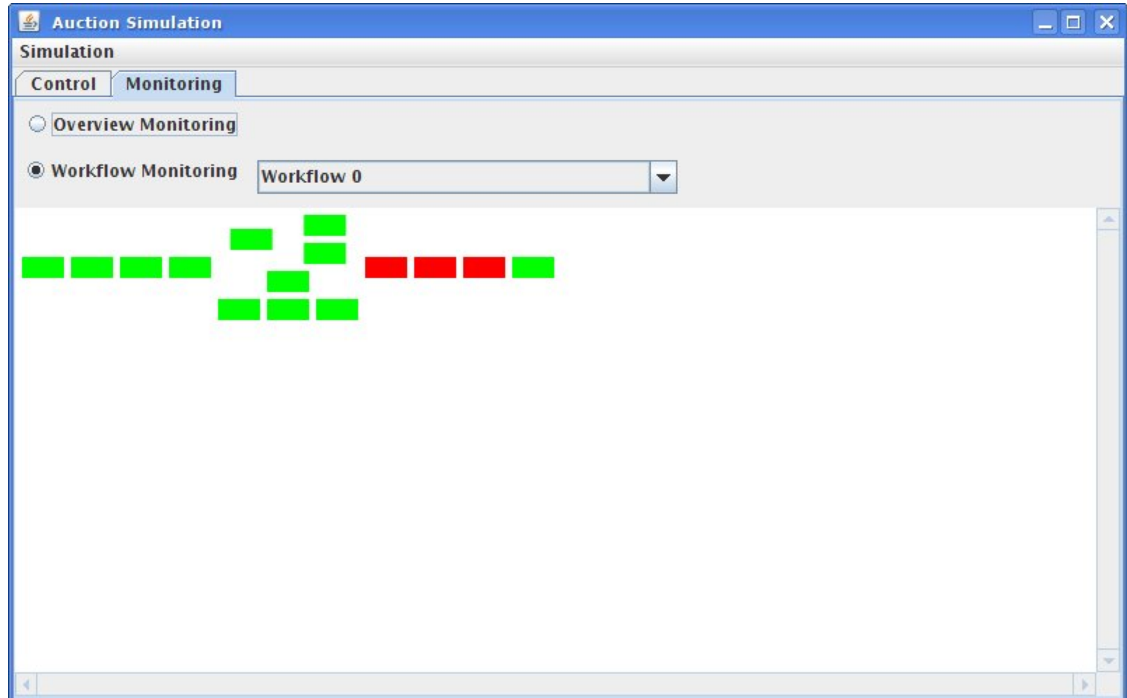

Abbildung 5.4: Die Überwachungsoberfläche des Simulations-Controllers.

5.6 Codestatistiken

Als Metrik für den Implementierungsaufwand wird die Anzahl der Java-Codezeilen für jeden größeren Programmteil und eine Gesamtzahl aufgeführt:

Programmteil	**Programmzeilen**
Converter	414
ServiceManager (inkl. Strategie)	3873 (1957)
SimulationController	3388
SimulationEnvironment	565
SimulationThreadPool	344
StandaloneService	397
Gesamt	**8981**

Tabelle 5.1: Codestatistik in Programmzeilen pro Programmteil.

Kapitel 6

Auswertung

Im folgenden Kapitel wird das SLO-Spiel (vgl. Abschnitt 3.5) durch Testläufe mit der Simulation und der verteilten Messung (vgl. Kapitel 4) nach den in Abschnitt 3.6 definierten Kriterien bewertet. Untersucht wird:

- Einschwingen auf einen stabilen Zustand.
- Verhalten unter dynamischer Last.
- Verhalten bei aktiviertem Scheduling.

Beim Einschwingvorgang wird davon ausgegangen, dass die SLO-Anteile für alle Käufer ausreichen. Bei der Untersuchung von dynamischer Last wird unterschieden ob das SLA in jedem Fall eingehalten wird oder ob es zeitweise verletzt wird.

Die Simulation wurde folgender Hardwareplattform durchgeführt:

- Intel Core 2 Duo 1.8 GHz
- 2 GB RAM

Bei allen Messungen wurde für eine Auktion 300 ms als Zeitdauer genutzt und 150 ms als Zeitdauer die ein Bieter auf Auktionen wartet (vgl. Abschnitt 4.2.3).

6.1 Bewertung des Einschwingvorgangs

Die Anzahl der Bieter ist gleichzusetzen mit der Anzahl der SLO-Verletzungen, d.h. je schneller alle Bieter beendet sind, desto besser ist das Ergebnis (vgl. Abschnitt 3.6).

Der Einschwingvorgang wird zuerst für Sequenzen ermittelt, anschließend für Workflows mit Verzweigungen. Untersucht wird der Einschwingvorgang mit folgenden Parametern:

- 10, 50 und 100 Agenten
- 50% der Agenten sind Verkäufer, 50% der Agenten sind Käufer.
- Das SLA kann in jedem Fall eingehalten werden.

Für 10 und 50 Agenten wurde ein Testlauf jeweils mit JGroups und dem JGroups-Ersatz vermessen. Für 10 Agenten zusätzlich ein Testlauf mit verteilten Agenten. Die Hardwareausstattung der einzelnen verteilten Agenten war:

- 6 Agenten identisch mit der Simulationsplattform
- 4 Agenten mit Pentium 4 und 1 GB RAM

Die verteilten Agenten wurden möglichst synchron, aber mit leichter Verzögerung zueinander gestartet (Die genaue Verzögerung wurde nicht ermittelt).

Auffällig ist hier, dass die Zeitdauer in der verteilten Messung (Abbildung 6.1) fast 10 mal so hoch ist, wie bei der Simulationsmessung mit JGroups (Abbildung 6.2). Erklärbar ist dieses Phänomen damit, dass in der verteilten Messung trotz höherer Rechenleistung für die einzelnen Agenten, die Latenzzeiten des Netzwerks eine entscheidende Rolle spielen. Dies ist auch daran zu sehen, dass der letzte Bieter mehrere Anläufe braucht, erfolgreich

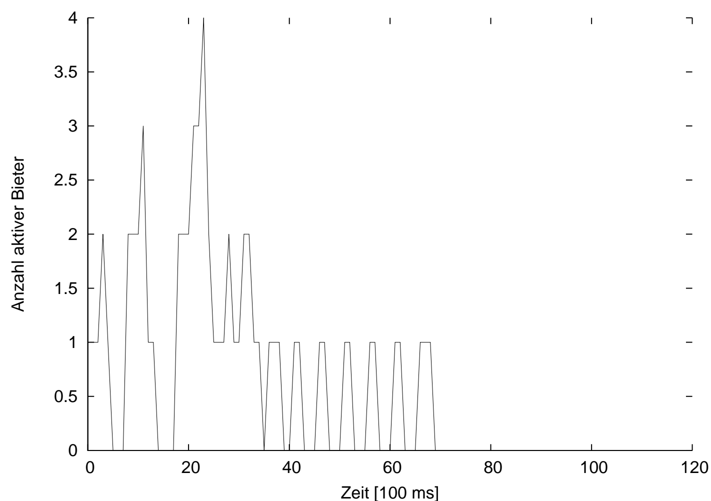

Abbildung 6.1: Einschwingverhalten aus Sicht der Bieter in der verteilten Messung mit 10 Agenten.

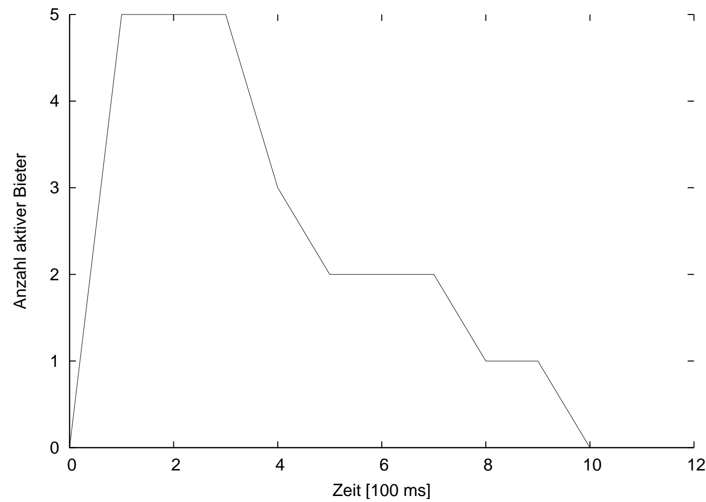

Abbildung 6.2: Einschwingverhalten aus Sicht der Bieter in der Simulation mit JGroups mit 10 Agenten.

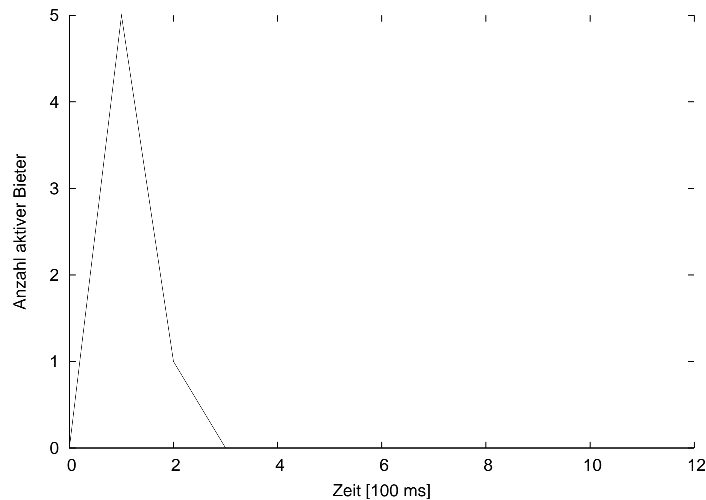

Abbildung 6.3: Einschwingverhalten aus Sicht der Bieter in der Simulation mit Ersatz für JGroups mit 10 Agenten.

an einer Auktion teilzunehmen. Das Einschwingverhalten im verteilten Fall unterscheidet sich zusätzlich durch den unterschiedlichen Start der Agenten. Da zu erst JGroups initialisiert und die entsprechende Sicht installiert werden muss, verzögert sich der Start der einzelnen Agenten. Dies ist zusätzlich abhängig von der Latenzzeit des Netzwerks.

Der Vergleich zwischen der Simulation mit JGroups (Abbildung 6.2) und dem Ersatz für JGroups (Abbildung 6.3) zeigt den erheblichen Aufwand der für die Kommunikation mit JGroups aufgewendet werden muss.

Weiterhin ist der Kommunikationsaufwand zu betrachten, je weniger Nachrichten über-

mittelt werden, desto besser ist der Mechanismus (vgl. Abschnitt 4.2.3).

Wie schon aus der Auswertung des Verhaltens der Bieter zu erwarten, steigt auch die Anzahl der Nachrichten deutlich an. Im verteilten Fall wird ein Spitzenwert von etwa 350 Nachrichten/s erreicht (Abbildung 6.4). Dies ist der in etwa $4,5$-fache Wert im Vergleich zum Simulationsmodell mit JGroups (Abbildung 6.5). Dies lässt sich ebenfalls mit der zu geringen Zeitdauer einer Auktion erklären. Je mehr Auktionen fehlschlagen, desto mehr Nachrichten werden generiert, da ein Bieter öfter Nachfragen an alle Agenten schicken muss.

Für die Auswertung der Simulation ohne JGroups wurde die Auflösung bei der Auswertung des Kommunikationsaufwands gesenkt (Abbildung 6.6). Da hier schon alle Auktionen nach etwa 300 ms beendet sind, ergibt die Auswertung des Kommunikationsaufwands sonst keinen Sinn. Man sieht hier deutlich, dass im Prinzip alle Nachrichten zu einem fast identischen Zeitpunkt verschickt werden. Dies entspricht auch der Erwartungshaltung nach der Auswertung des Verhaltens der Bieter.

Betrachtet werden nun weitere Ergebnisse mit 50 bzw. 100 Agenten. Prinzipiell bestätigen die Ergebnisse (Abbildung 6.7 und Abbildung 6.8) die Tendenz, dass die Simulation mit JGroups im vergleich zur Simulation ohne JGroups deutlich schlechter abschneidet. Allerdings findet die Simulation aus bisher nicht ersichtlichen Gründen trotz dem exakt gleichen Service- und Workflowmodell keine Lösung.

Auch beim Kommunikationsaufwand bestätigt sich das Ergebnis. In der Messung ohne JGroups (Abbilding 6.10) werden fast alle Nachrichten sehr schnell, fast zum gleichen

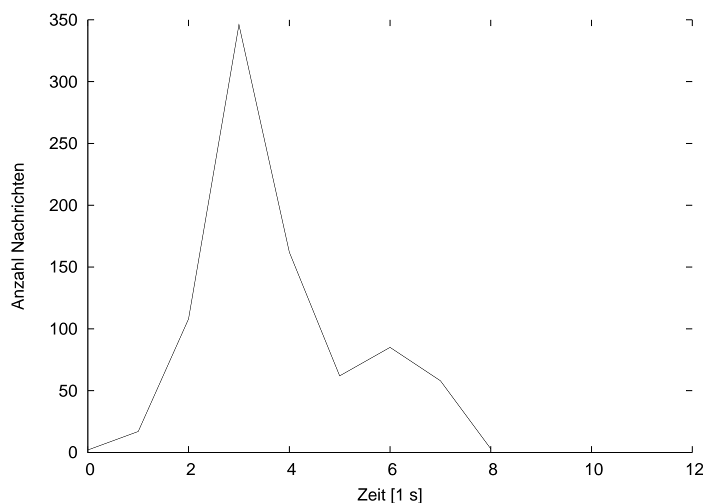

Abbildung 6.4: Der Kommunikationsaufwand des Einschwingverhaltens in der verteilten Messung mit 10 Agenten.

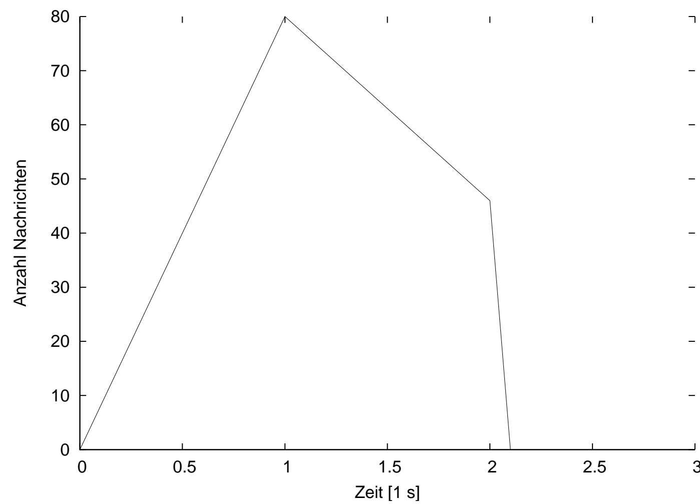

Abbildung 6.5: Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit JGroups mit 10 Agenten.

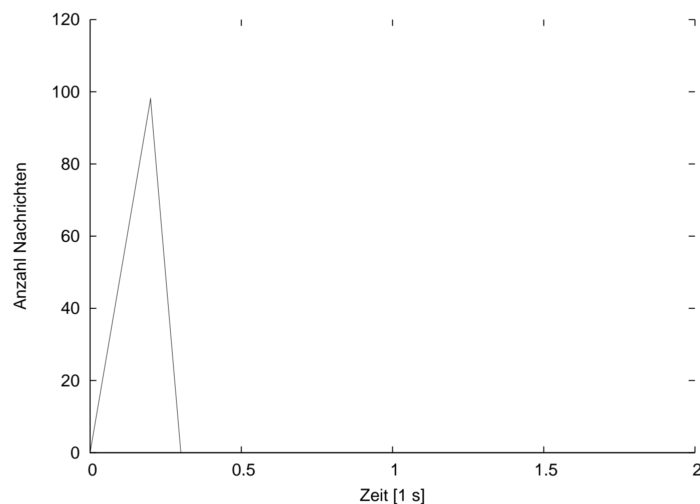

Abbildung 6.6: Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit Ersatz für JGroups mit 10 Agenten.

Zeitpunkt verschickt. In der Messung mit JGroups (Abbilding 6.9) zeigt sich ein deutlich geringerer Spitzendurchsatz an Nachrichten, dies scheint eine Limitierung durch JGroups zu sein, da die 25 Agent, welche in diesem Szenario Bieter sind, auch in dieser Messung schnell erkannt werden. Zusätzlich zeigt sich, dass sobald kein Ergebnis gefunden wird der Kommunikationsaufwand des SLO-Spiels durch kontinuierliche Multicasts deutlich erhöht ist.

Die Auswertung der Ergebnisse mit 100 Agenten, diesmal nur ohne JGroups zeigt, wann auch diese Lösung an eine Limitierung stößt. Sie findet zwar immernoch eine Lösung,

zeigt aber, dass die Rechenzeit des Simulationsrechners nicht ausreicht, um alle Agenten parallel zu berechnen. Nachdem die Bieter feststellen, dass sie aus unterschiedlichen Gründen keine passende Auktion finden, kehren sie zum Wartezustand zurück. Im Normalfall sollte sehr schnell erkannt werden, dass immernoch eine SLO-Verletzung vorliegt und der Agent erneut zum Bieter werden.

Mit höheren Ablaufzeiten für die Auktion und Wartezeiten für die Bieter könnte hier eventuell ein besseres Ergebnis erreicht werden. Dies hätte zur Folge, dass weniger Nachrichten versendet werden, da Bieter länger warten und zusätzlich mehr Bieter früher erfolg-

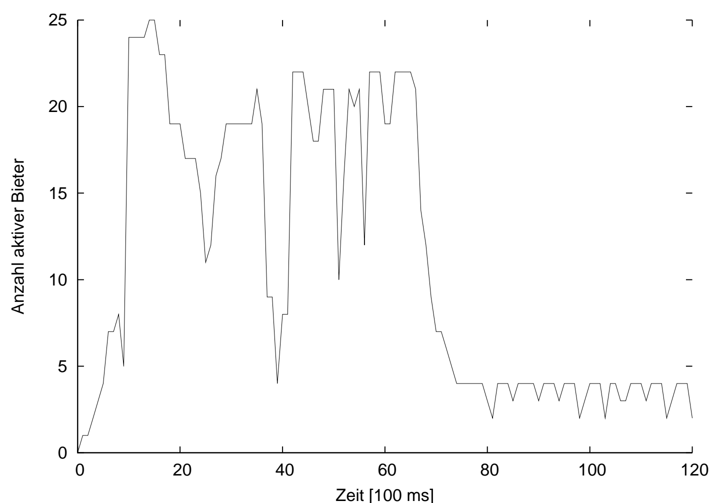

Abbildung 6.7: Einschwingverhalten aus Sicht der Bieter in der Simulation mit JGroups mit 50 Agenten.

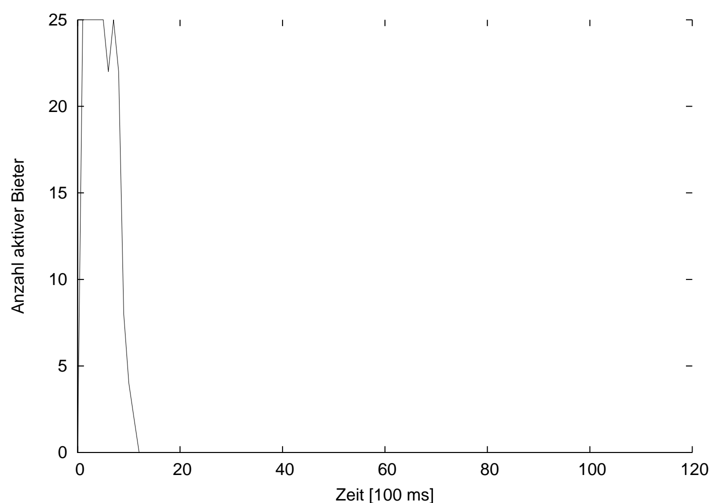

Abbildung 6.8: Einschwingverhalten aus Sicht der Bieter in der Simulation mit Ersatz für JGroups mit 50 Agenten.

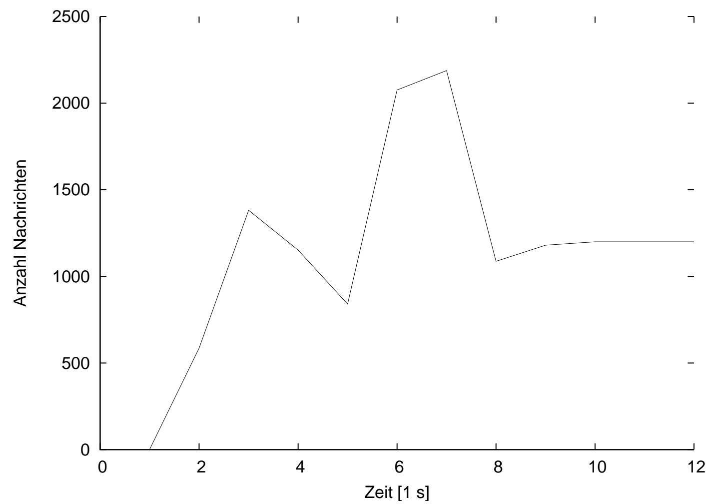

Abbildung 6.9: Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit JGroups mit 50 Agenten.

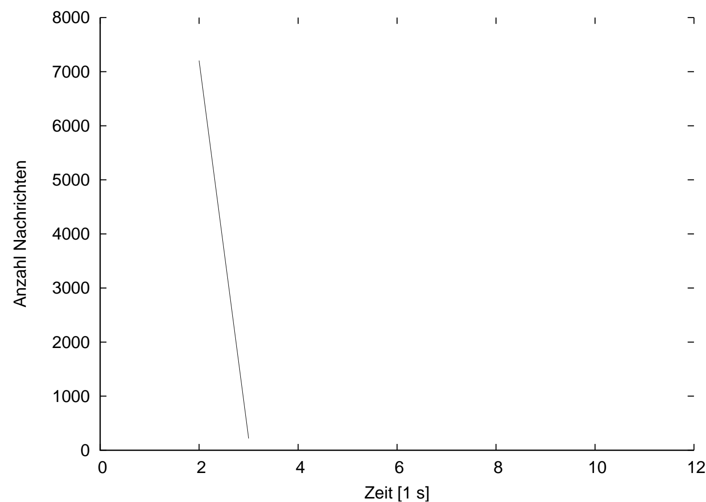

Abbildung 6.10: Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit Ersatz für JGroups mit 50 Agenten.

reich an einer Auktion teilnehmen könnten, denn wertet man die Anzahl der Auktionen über den Zeitraum des Simulationslaufs aus (Abbildung 6.13), lässt sich erkennen, dass viele Auktionen erfolglos abbrechen.

Bisher wurden der Einschwingvorgang in Sequenzen ausgewertet, dies kann auch für Workflows mit zusammengesetzten Aktivitäten ermittelt werden.

Es zeigt sich, dass beim Vorhandensein von mehreren zusammengesetzten Aktivitäten im Vergleich zu 10 Agenten in einer Sequenz kein wesentlicher Unterschied erscheint. Ein zeitlicher Unterschied wäre zu erwarten, da die Vertreter in einer zusammengesetzten

Aktivität sich mit den enthaltenen Aktivitäten synchronisieren müssen, da sonst Race-Conditions entstehen.

Mit nur insgesamt 36 Nachrichten kann hier schon eine Lösung gefunden werden. Dies folgt aus wesentlich kleineren Gruppengrößen, da die Nachrichtenanzahl eines Multicast sich mit der Anzahl der Agenten in einer Gruppe linear erhöht.

Als Ergebnis der Auswertung des Einschwingvorgangs lassen sich mehrere Dinge festhalten:

- Für ein reales System muss eine Auktion länger andauern, da durch Netzwerklaten-

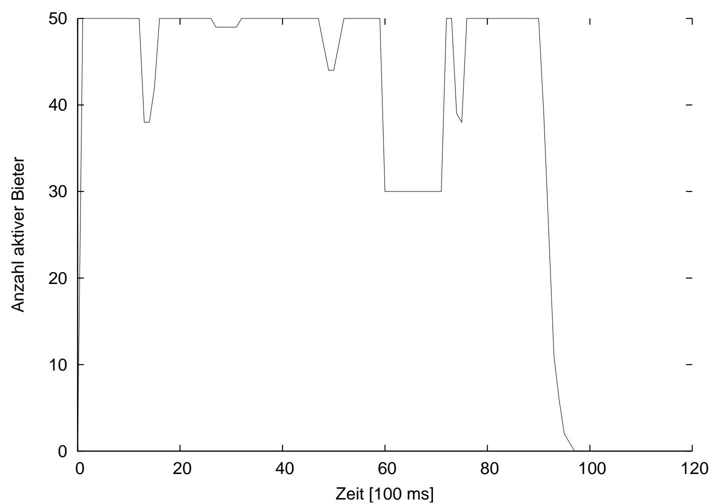

Abbildung 6.11: Einschwingverhalten aus Sicht der Bieter in der Simulation mit Ersatz für JGroups mit 100 Agenten.

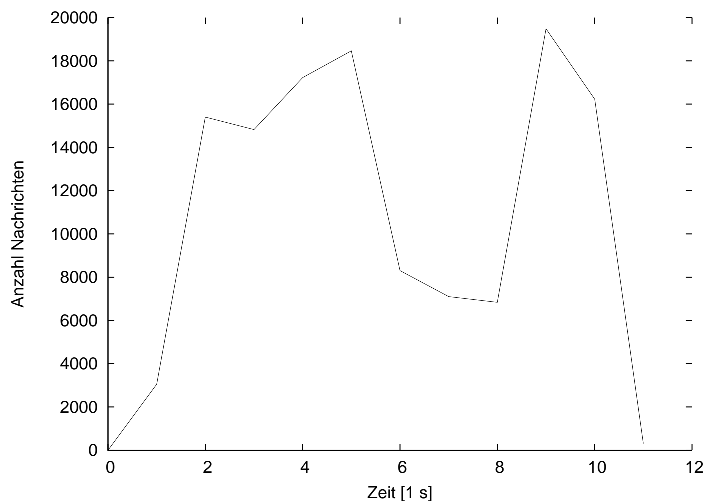

Abbildung 6.12: Der Kommunikationsaufwand des Einschwingverhaltens in der Simulation mit Ersatz für JGroups mit 100 Agenten.

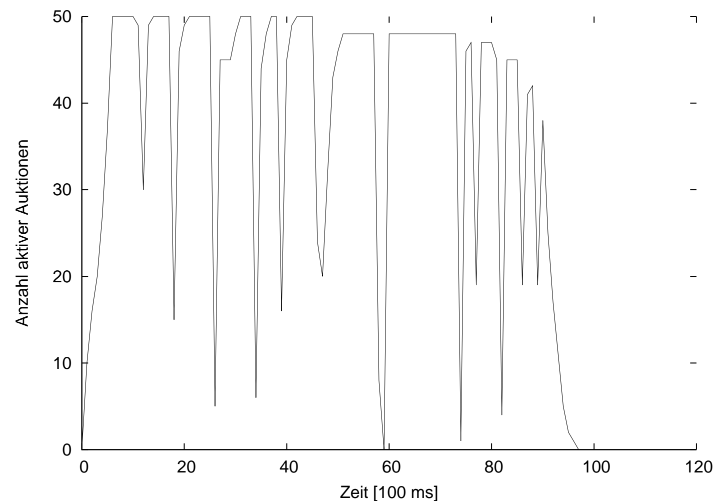

Abbildung 6.13: Anzahl der Auktionen über die Zeit in der Simulation mit Ersatz für JGroups mit 100 Agenten.

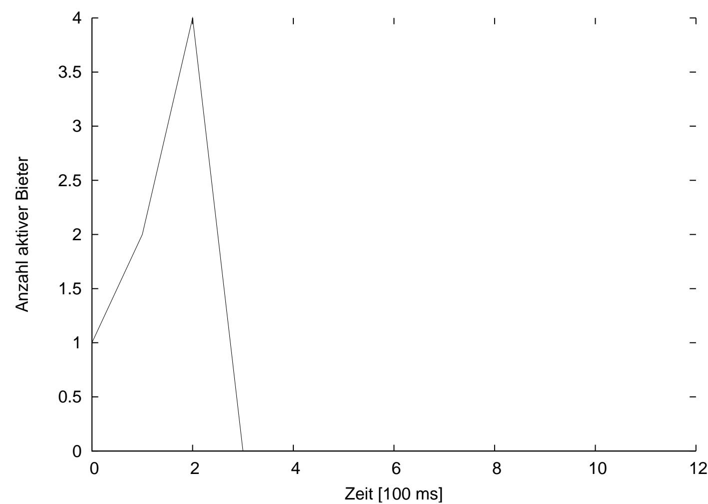

Abbildung 6.14: Einschwingverhalten aus Sicht der Bieter in der Simulation mit Ersatz für JGroups mit 10 Agenten und mehreren zusammengesetzten Aktivitäten.

zen viele Bieter keine passenden Auktionen finden.

- Ein reales System wird deshalb zumindest für wenige Agenten langsamer als die Simulations-Umgebung sein.

- Durch JGroups entsteht ein erheblicher Mehraufwand in der Simulation.

- Ab etwa 100 Agenten kommt das SLO-Spiel an Kapazitätsgrenzen.

- Zusammengesetzte Aktivitäten verändern das Ergebnis nicht wesentlich.

6.2 Verhalten während dynamischer Änderungen

Das SLO-Spiel sollte möglichst schnell auf dynamische Änderungen im System reagieren können, so dass die Zeitdauer von SLO-Verletzungen minimiert wird. Für die Auswertung dynamischer Änderungen wird Lastmodell verwendet das ähnlich zu dem in [TTU06] genutzten Lastmodell ist:

- Die initialen Antwortzeiten unterscheiden sich zwischen den Services.
- Nach einer kurzen Wartezeit zu Beginn des Simulationslaufs wird von jedem Service für eine kurz zufällige Zeitdauer die Antwortzeit um einen zufälligen Wert erhöht.
- Zu einem festgelegten Zeitpunkt fällt die Antwortzeit wieder auf das ursprüngliche Niveau ab, um anschließend nach einer zufälligen Zeitdauer wieder anzusteigen.

Es werden 10 und 100 Agenten betrachtet. Alle Messungen wurden mit der Simulations-Umgebung und dem Ersatz für JGroups durchgeführt. Die Zeitdauer der Simulation beträgt 200 Sekunden.

Es soll nun untersucht werden, wie sich das System verhält, wenn:

- durch die Änderungen die Summe der Antwortzeiten entlang aller Pfade eines Workflow immer kleiner oder gleich wie der SLA-Zielwert ist und

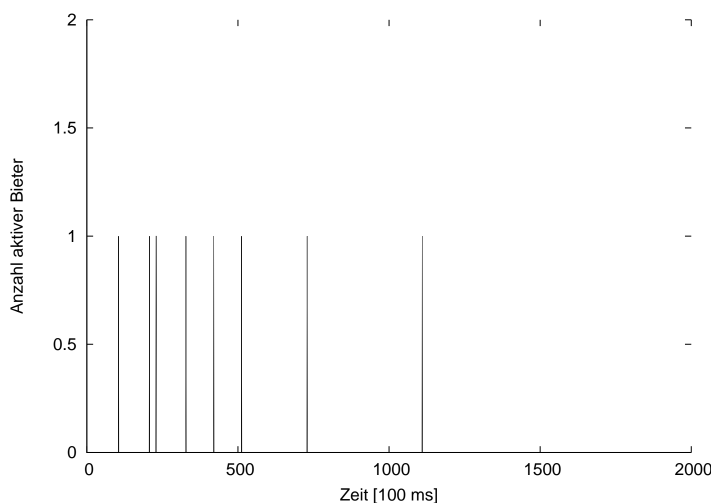

Abbildung 6.15: Verhalten der Bieter im SLO-Spiels bei dynamischen Änderungen und 10 Agenten.

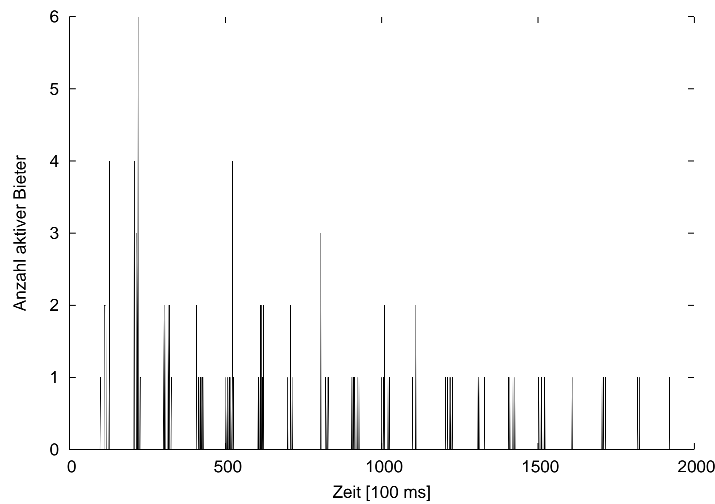

Abbildung 6.16: Verhalten der Bieter im SLO-Spiels bei dynamischen Änderungen und 100 Agenten.

- durch die Änderungen die Summe der Antwortzeiten entlang eines Pfads größer als der SLA-Zielwert sein kann.

Zuerst wird der erste Fall ausgewertet.

Eine grundsätzliche Tendenz ist, dass das SLO-Spiel es offenbar schafft, im Verlauf der Simulation das System so zu verbessern, dass trotz gleichem Lastmodell weniger SLO-Verletzungen auftreten. Während am Anfang vor allem bei 100 Agenten deutlich mehr Agenten zu Bietern werden, reduziert sich dies im weiteren Verlauf (Abbildung 6.16).

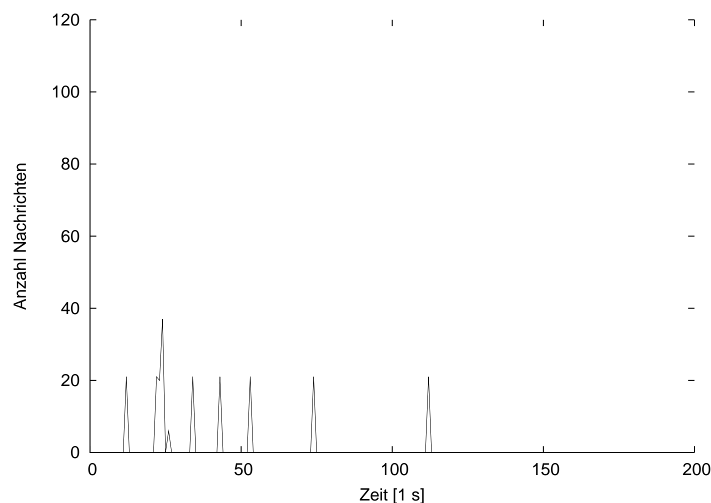

Abbildung 6.17: Kommunikationsaufwand des SLO-Spiels bei dynamischen Änderungen und 10 Agenten.

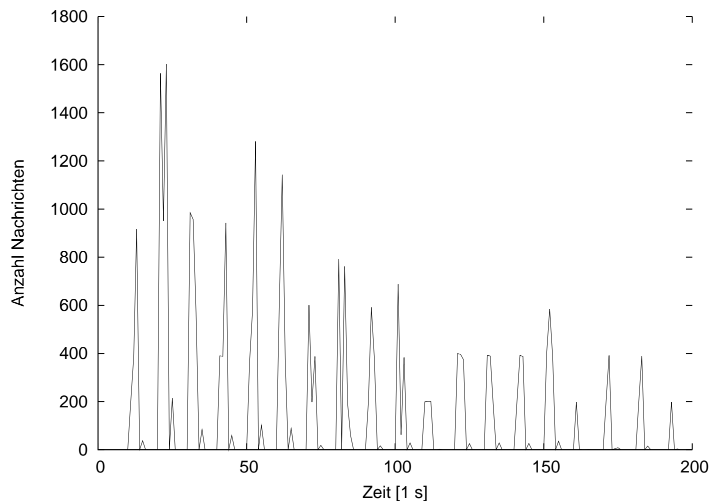

Abbildung 6.18: Kommunikationsaufwand des SLO-Spiels bei dynamischen Änderungen und 100 Agenten.

Dieses Verhalten ist im Prinzip auch für den anderen Versuch erkennbar (Abbildung 6.15).

Auch der Verlauf der Nachrichten ist bei 100 Agenten rückläufig (Abbildung 6.18), dies ist bei einer reduzierten Anzahl von Bieter zu erwarten. Für Bieter 1 werden wie erwartet keine Nachrichten mehr gesendet, nachdem kein Agent mehr zum Bieter wird (Abbildung 6.17).

Ein weiterer Interessanter Aspekt des Kommunikationsaufwand ist, dass bei einer nur maximal 3-fachen Menge an Bietern in der Spitze die 40-fache Menge an Nachrichten beim Versuch mit 100 Agenten versendet wird.

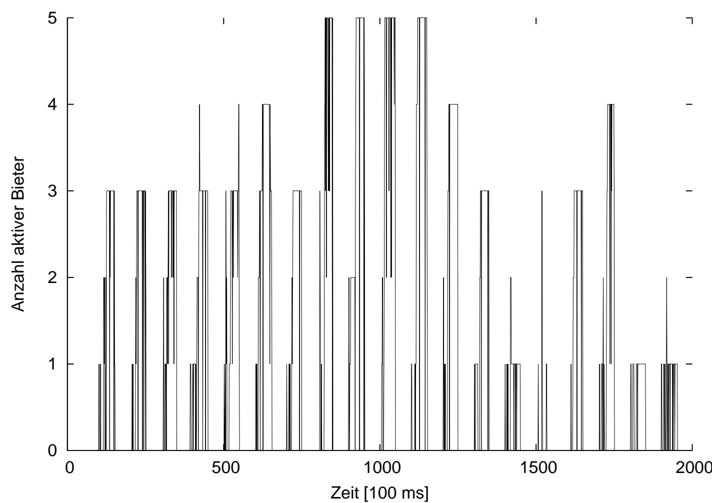

Abbildung 6.19: Verhalten der Bieter im SLO-Spiels bei dynamischen Änderungen und 10 Agenten bei Überschreitung des Workflow SLAs.

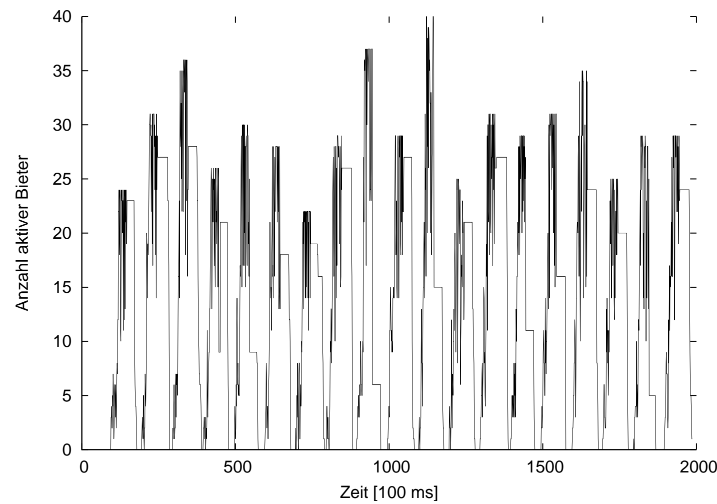

Abbildung 6.20: Verhalten der Bieter im SLO-Spiels bei dynamischen Änderungen und 100 Agenten bei Überschreitung des Workflow SLAs.

Wird das Workflow-SLA überschritten, sollte das SLO-Spiel sich weiterhin gutmütig verhalten, d.h. wenn möglich zu einer Besserung der Situation beitragen und auf keinen Fall die Situation verschlechtern.

Wie in den Abbildungen 6.19 und 6.20 zu sehen ist, erkennt das SLO-Spiel eine Reihe von SLO-Verletzungen, denn Agenten werden zu Bietern. Kaum ein Bieter findet allerdings eine Auktion.

Stattdessen wird ein erheblicher Kommunikationsmehraufwand erzeugt wie in den Abbildungen 6.21 und 6.22 zu sehen ist. Wird für jede versendete Nachricht im Durchschnitt

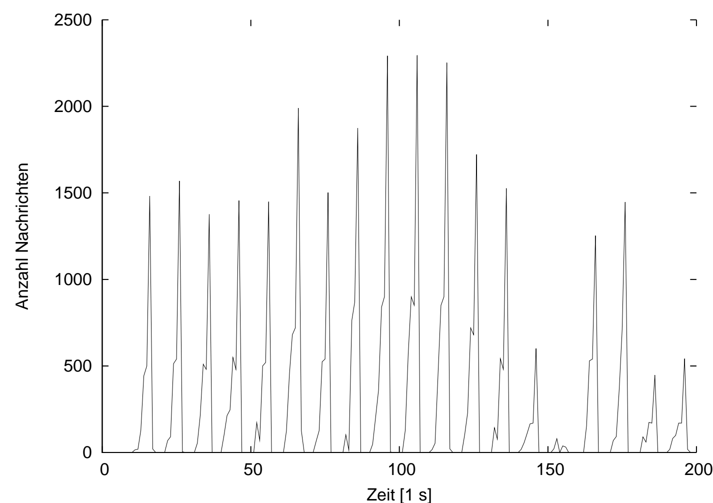

Abbildung 6.21: Kommunikationsaufwand des SLO-Spiels bei dynamischen Änderungen und 10 Agenten bei Überschreitung des Workflow SLAs.

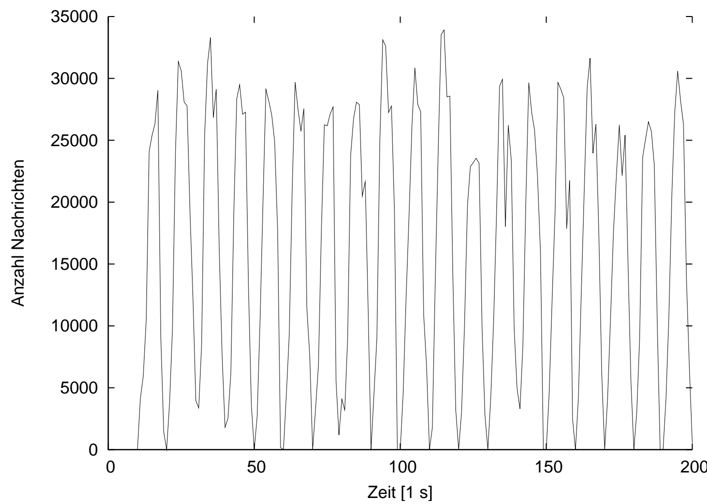

Abbildung 6.22: Kommunikationsaufwand des SLO-Spiels bei dynamischen Änderungen und 100 Agenten bei Überschreitung des Workflow SLAs.

512 Bytes angenommen errechnet sich eine theoretische Spitzenlast von etwa 130 MBit/s bei 100 Knoten. Dies wird vermutlich ein reales System endgültig zum Erliegen bringen.

Fazit der Auswertung ist, dass solange das SLA eingehalten wird eine Verbesserung erreicht werden kann. Sobald keine Lösung ermittelt werden kann, da die Summe der Antwortzeiten aller Pfade des Workflows das SLA überschreiten, erhöht sich der Kommunikationsaufwand mit jeder SLO-Verletzung drastisch. Dies ist nach der theoretischen Betrachtung in Abschnitt 3.5.3 zu erwarten.

Eventuell ist als Lösung dieses Problems eine Umschaltung zwischen den beiden in Abschnitt 3.5.3 betrachteten Benachrichtigungsmechanismen für dezentrale Auktionen möglich. Dazu müssten geeignete Vorkehrungen getroffen werden, um zu Erkennen wann zwischen den einzelnen Benachrichtigungsmodi umgeschaltet werden soll.

6.3 Verhalten bei aktiviertem Scheduling

Das in Abschnitt 3.5.4 beschriebene Scheduling verschiebt Antwortzeitanteile zwischen Aktivitäten die von einem einzigen Service ausgeführt werden. Die grundsätzliche Idee ist, dass dadurch Verbesserungen erreicht werden können, die durch reines Verschieben von SLO-Anteilen innerhalb eines Workflows nicht möglich sind.

Die theoretisches Analyse der Erweiterung des SLO-Spiels führt aber zum Ergebnis, dass diese Erweiterung auch zu Verschlechterungen des Ergebnisses führen kann (vgl. Abschnitt 3.5.4). Dies gilt es praktisch zu überprüfen.

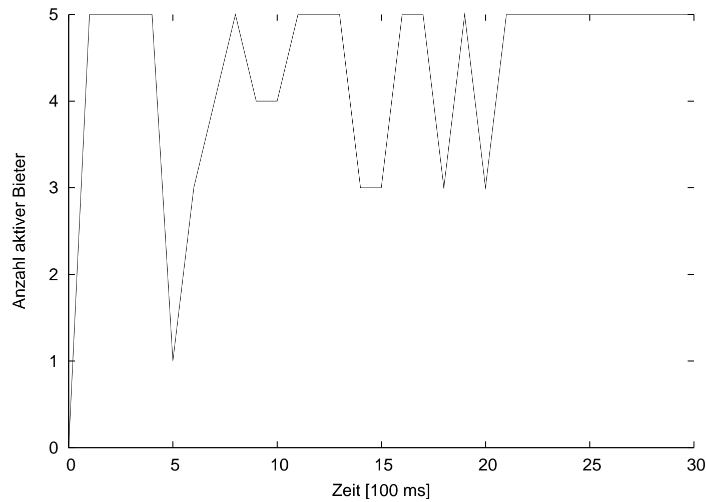

Abbildung 6.23: Verhalten der Bieter des Workflows mit dem niedrigeren Wert.

Der Versuchsaufbau sieht folgendermaßen aus:

- Es gibt zwei Workflows mit unterschiedlichem Wert (d.h. es gibt mehr oder weniger virtuelles Geld zum Bieten), bestehend aus 10 Agenten in einer Sequenz
- Aktivitäten der beiden Workflows teilen sich einige Services
- Es kann nur das SLA des Workflows mit dem niedrigeren Wert tatsächlich eingehalten werden.
- Die Antwortzeit aller simulierten Services wird während der Simulationslaufzeit nicht manipuliert.

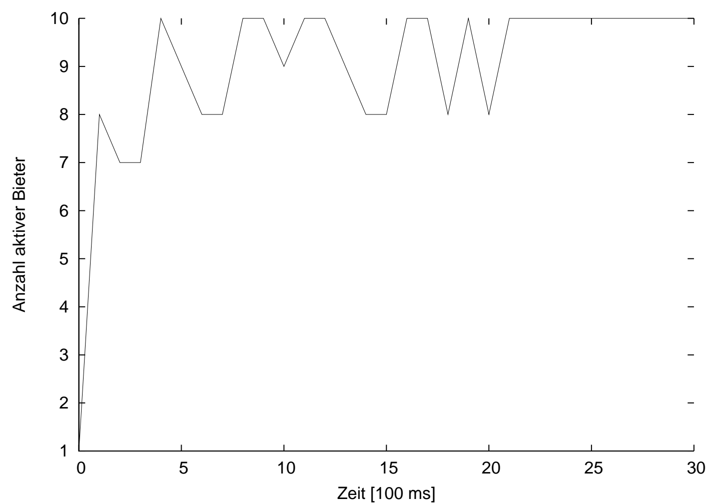

Abbildung 6.24: Verhalten der Bieter des Workflows mit dem höheren Wert.

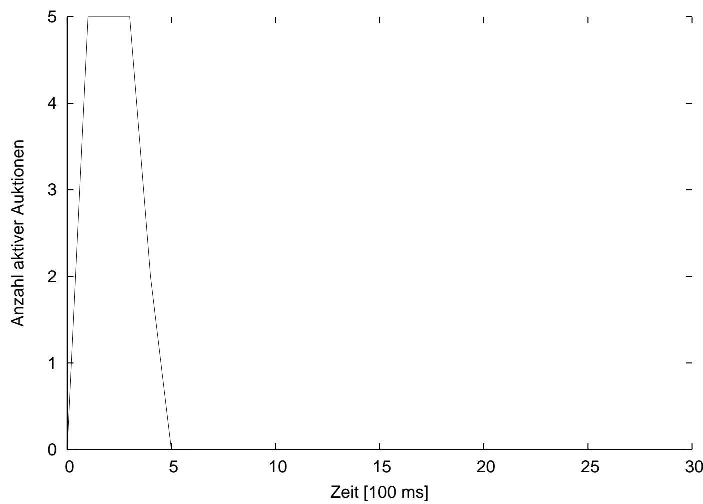

Abbildung 6.25: Anzahl durchgeführter Auktionen pro Zeit bei aktiviertem Scheduling.

Es stellt sich heraus, dass das theoretische Ergebnis auch in der Praxis erreicht wird. Es kann also mittels dieser Erweiterung des SLO-Spiels zwar zu Verbesserungen kommen, wenn beispielsweise beide SLAs eingehalten werden können, sobald eine gewisse Menge an Antwortzeitanteilen verschoben wurde, dies muss allerdings nicht zwangsläufig so sein.

Die Abbildungen 6.23 und 6.24 stellen dar, wie sich die Agenten der unterschiedlichen Workflows verhalten. Abbildung 6.25 zeigt, dass tatsächlich Auktionen durchgeführt werden, allerdings werden alle Anteile vom Workflow mit dem höheren Wert aufgekauft.

Kapitel 7

Zusammenfassung

Das Ziel der Arbeit war die Entwicklung eines selbstorganisierenden Systems, das durch geeignete Management-Aktivitäten Dienstgüteeigenschaften in einem Service-orientierten System verbessern kann. Der Fokus der Entwicklung lag auf der Entwicklung eines sogenannten *SLO-Spiels*, welches mit Hilfe des aus der Spieltheorie stammenden algorithmischen Mechanismus-Design entworfen wurde.

Dazu wurden zuerst mögliche Eigenschaften eines Mechanismus definiert (Abschnitt 2.3) und die Klasse der Vickrey-Clarke-Groves-Mechanismen (VCG) vorgestellt (Abschnitt 2.3.8). VCG-Mechanismen mit der der sogenannten Clarke-Pivot-Regel besitzen einige gewünschte Eigenschaften:

- *Maximierung des sozialen Wohls*, d.h. das durch den Mechanismus ermittelte Ergebnis ist ein globales Optimum.
- *Individuelle Rationalität*, d.h. jeder Agent zieht durch die Teilnahme am Mechanismus mehr Nutzen, als wenn er nicht daran teilnehmen würde.
- *Anreizkompatibilität*, d.h. jeder Agent wird dazu angereizt eine wahrheitsgemäße Strategie zu spielen.

Die anschließend vorgestellte Vickrey-Auktion ist genau ein solcher VCG-Mechanismus mit Clarke-Pivot-Regel und hat deswegen die gleichen Eigenschaften. Weiterhin besitzt die Vickrey-Auktion eine niedrige Kommunikationskomplexität.

Weiterhin wurde in Abschnitt 3.2 die Struktur eines Service-orientierten Systems definiert und SLM-Eigenschaften dafür festgelegt (Abschnitt 3.3). Zur Vereinfachung des

Problems wurde im weiteren Verlauf die Antwortzeit als einziger Dienstgüteparameter betrachtet.

Für das Service-orientierte System wurden zwei prinzipielle Möglichkeiten gefunden Verletzungen der maximalen Antwortzeit zu beheben:

- Lockerung der Antwortzeitvorgabe, bei gleichzeitiger Straffung dieser Vorgabe an anderer Stelle.
- Erhöhung der Ressourcen für einen Service, so dass die Antwortzeit sinkt.

Für dieses System wurde anschließend auf Basis des Mechanismus-Designs das SLO-Spiel entwickelt. Das SLO-Spiel betrachtet zunächst ein vereinfachtes Modell und als Management-Aktivität nur die Lockerung der Antwortzeitvorgaben. Das entwickelte SLO-Spiel ist ein Mechanismus mit folgenden Eigenschaften:

- Durch einen Service werden Aktivitäten ausgeführt, für jede Aktivität gibt es einen Agenten im SLO-Spiel.
- Kann ein Agent Antwortzeitanteile abgeben, wird er zum Auktionator.
- Benötigt ein Agent Antwortzeitanteile, wird er zum Bieter.
- Jede Auktion wird verteilt durch einen eigenen Auktionator mit den Regeln der Vickrey-Auktion durchgeführt.

Die Wahl der Vickrey-Auktion fiel auf Grund verschiedener theoretischer Überlegungen. Zum einen hat Vickrey in [Vic61] bewiesen, dass in bestimmten Fällen der Ertrag des Verkäufers für andere Auktionsformen, z.B. die englische Auktion gleich hoch ist. Außerdem benötigt die Vickrey-Auktion nur wenige Kommunikationsschritte, um zu einem Ergebnis zu kommen.

Auf Grund des Aufbaus des vereinfachten SLO-Spiels konnten in dieser Arbeit verschiedene theoretische Eigenschaften bewiesen werden:

- Das vereinfachte SLO-Spiel erreicht nach einer endlichen Anzahl von Auktionen immer ein effizientes Ergebnis, sofern ein Ergebnis existiert.
- Das SLO-Spiel ist für alle Agenten individuell rational, da jede einzelne Auktion individuell rational ist.

- Das SLO-Spiel ist budget-ausgeglichen, da weder Geld in das System fließt und auch kein Geld aus dem System abfließt.
- Agenten des SLO-Spiels sind prinzipiell ungeduldig, deshalb wird von den Agenten weiterhin das dominante Gleichgewicht der Vickrey-Auktion gespielt.

Für die verallgemeinerte Variante des SLO-Spiels gelten nicht alle Eigenschaften:

- Das verallgemeinerte SLO-Spiel ist nicht individuell rational
- Das verallgemeinerte SLO-Spiel ist nicht effizient, d.h. es wird nicht immer ein effizientes Ergebnis gefunden.

Im Anschluss daran wurde die Umsetzung des SLO-Spiels entworfen und implementiert, damit Leistungsmessungen des SLO-Spiels gemacht werden können. Dabei wurde eine Simulations-Umgebung, als auch eine verteilte Umgebung entwickelt, damit auch über einige Knoten verteilte Messungen gemacht werden konnten. Das Kernstück der Umsetzung ist eine Strategie, welche das SLO-Spiel implementiert.

Die Ergebnisse der Leistungsmessungen werden hier kurz zusammengefasst:

- Das vereinfachte SLO-Spiel findet im Allgemeinen ein effizientes Ergebnis, auch in der verteilten Umgebung.
- Das vereinfachte SLO-Spiel verbessert das Ergebnis bei dynamischer Last, sofern die Gesamtantwortzeit kleiner als das Vorgabe für den gesamten Workflow ist.
- Ist die Gesamtantwortzeit größer als die Vorgabe für den gesamten Workflow, erzeugt das SLO-Spiel einen sehr großen Kommunikationsaufwand.
- Im verallgemeinerten SLO-Spiel mit mehreren Workflows, gibt es Situationen in denen das SLO-Spiel die Situation verschlechtert, anstatt verbessert.

Die Simulationen deuten ebenfalls an, dass bei etwa 100 Agenten die Leistung des SLO-Spiels bei den gewählten Parameter in der Simulation an Lastgrenzen stößt, so dass die Skalierbarkeit über 100 Agenten nicht gegeben ist.

In der theoretischen Analyse und durch die Leistungsmessungen konnten verschiedene Probleme des Ansatzes festgestellt werden, die in weiteren Arbeiten bearbeitet werden können. Die wichtigsten theoretischen Probleme sind:

- Das verwendete Scheduling-Modell ist stark vereinfacht. Eine Verbesserung hierfür wäre ein Scheduling-Modell, das realistischere Annahmen macht.

- Das verallgemeinerte SLO-Spiel ist nicht effizient. Da jeder Agent nur lokale Informationen besitzt, kann dieser keine bessere Entscheidung treffen. Es müsste also eine geeignete Kommunikationsstruktur entworfen werden, um Informationen zwischen den Agenten zu tauschen.

- Sowohl Agenten als auch unterlagerte Netzwerke können ausfallen. Das SLO-Spiel sollte möglichst tolerant mit Ausfällen umgehen können. Es sollte also analysiert werden, wie das SLO-Spiel mit Ausfällen umgehen sollte.

Aus der Leistungsbewertung folgt ein weiteres offenes Problem:

- Der Kommunikationsaufwand steigt bei SLA-Verletzungen enorm an. Dies würde in der Realität zu einer weiteren Verschlechterung der Leistung des Systems führen. Das Hauptproblem ist die Bekanntmachung von Auktionen im verteilten Fall. Hier müsste ein besseres Kommunikationsschema gefunden werden oder alternative Kommunikationswege in Betracht gezogen werden.

Prinzipiell ist weder die Umsetzung des vereinfachten noch des verallgemeinerten SLO-Spiels bisher praxistauglich. Beide Varianten erzeugen bei grundsätzlich schon kritischen Lasten erhebliche zusätzliche Last. Auch eine Umkehrung des Kommunikationswegs wie es in Abschnitt 3.5.3 auch betrachtet wird, kehrt die Situation um, d.h. die durch das SLO-Spiel erzeugte Last steigt an, je mehr Agenten ihre Antwortzeitvorgaben einhalten.

Das verallgemeinerte SLO-Spiel hat zusätzlich das entscheidende Problem, dass in einigen Situationen durch die Management-Aktivitäten die Dienstgüte sogar reduziert wird. Dies zeigt, dass hier noch erheblicher Weiterentwicklungsbedarf besteht.

Anhang A

Literaturverzeichnis

[AC04] AUSUBEL, Lawrence M. ; CRAMTON, Peter: Vickrey Auctions with Reserve Pricing. 2004. – Forschungsbericht

[ACKM04] ALONSO, Gustavo ; CASATI, Fabio ; KUNO, Harumi ; MACHIRAJU, Vijay: *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004. – ISBN 3–540–44008–9

[Afe06] AFERGAN, Michael: Using Repeated Games to Design Incentive-Based Routing Systems. In: *INFOCOM*, 2006

[Apa08] Apache Foundation: *Xerces Version 2.9.1*. `http://xml.apache.org/xerces2-j/index.html`. Version: 2008

[AS92] AUMANN, Robert J. ; SHAPLEY, Lloyd S.: Long Term Competition-A Game Theoretic Analysis / UCLA Department of Economics. 1992 (676). – UCLA Economics Working Papers

[Ban98] BAN, Bela: Design and Implementation of a Reliable Group Communication Toolkit for Java. 1998. – Forschungsbericht

[Ban08a] BAN, Bela: *JGroups JavaDoc Dokumentation*. `http://www.jgroups.org/javagroupsnew/docs/javadoc/index.html`. Version: September 2008

[Ban08b] BAN, Bela: *Reliable Multicasting with the JGroups Toolkit*. `http://www.jgroups.org/javagroupsnew/docs/manual/html/`. Version: January 2008

[BM04] BIRON, Paul V. (Hrsg.) ; MALHOTRA, Ashok (Hrsg.): *XML Schema Part 2: Datatypes*. Second. W3C, 2004 (W3C Recommendation)

[Bra00] BRANDT, Felix: Antisocial Bidding in Repeated Vickrey Auctions / Technische Universität München. 2000. – Forschungsbericht

[BW01] BRANDT, Felix ; WEISS, Gerhard: Antisocial Agents and Vickrey Auctions. In: MEYER, John J. (Hrsg.) ; TAMBE, Milind (Hrsg.): *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, 2001, S. 120–132

[CHTCB96] CHANDRA, Tushar D. ; HADZILACOS, Vassos ; TOUEG, Sam ; CHARRON-BOST, Bernadette: On the impossibility of group membership. In: *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA : ACM, 1996. – ISBN 0–89791–800–2, S. 322–330

[CKV01] CHOCKLER, Gregory V. ; KEIDAR, Idid ; VITENBERG, Roman: Group communication specifications: a comprehensive study. In: *ACM Comput. Surv.* 33 (2001), Dezember, Nr. 4, S. 427–469. – ISSN 0360–0300

[CS03] CONITZER, Vincent ; SANDHOLM, Tuomas: *Complexity Results about Nash Equilibria*. 2003. – paper

[CT96] CHANDRA, Tushar D. ; TOUEG, Sam: Unreliable failure detectors for reliable distributed systems. In: *J. ACM* 43 (1996), Nr. 2, S. 225–267. – ISSN 0004–5411

[DAGP90] DENEUBOURG, Jean-Louis ; ARON, Serge ; GOSS, Simon ; PASTEELS, Jacques M.: The self-organizing exploratory pattern of the Argentine ant. In: *Journal of Insect Behavior* 3 (1990), März, Nr. 2, S. 159–168

[DB05] DORIGO, Marco ; BLUM, Christian: Ant Colony Optimization Theory: A Survey. In: *Theoretical Computer Science* 344 (2005), Nr. 2–3, S. 243–278

[DDK+04] DAN, Asit ; DAVIS, Doug ; KEARNEY, Robert ; KELLER, Alexander ; KING, Richard P. ; KUEBLER, Dietmar ; LUDWIG, Heiko ; POLAN, Mike ; SPREITZER, Mike ; YOUSSEF, Alaa: Web services on demand: WSLA-driven automated management. In: *IBM Systems Journal* 43 (2004), Nr. 1, S. 136–158

[Deb04] DEBUSMANN, Markus: *Modellbasiertes Service Level Management verteilter Anwendungssysteme*, Universität Kassel, Diss., Dezember 2004

[DG97] DORIGO, Marco ; GAMBARDELLA, Luca M.: Ant colony system: A cooperative learning approach to the traveling salesman problem. In: *IEEE Transactions on Evolutionary Computation* 1 (1997), S. 53–66

[DGP06] DASKALAKIS, Constantinos ; GOLDBERG, Paul W. ; PAPADIMITRIOU, Christos H.: The complexity of computing a Nash equilibrium, ACM Press, 2006, S. 71–78

[DJP03] DASH, Rajdeep K. ; JENNINGS, Nicholas R. ; PARKES, David C.: Computational-Mechanism Design: A Call to Arms. In: *IEEE Intelligent Systems* 18 (2003), Nr. 6, S. 40–47. – ISSN 1541–1672

[EK02] EMMERICH, W. ; KAVEH, N.: Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model, 691–692

[FL01] FLEURY, Marc ; LINDFORS, Juha: *JMX: Managing J2EE Applications with Java Management Extensions*. Indianapolis, IN, USA : Sams, 2001. – ISBN 0672322889

[FLM94] FUDENBERG, Drew ; LEVINE, David I. ; MASKIN, Eric: The Folk Theorem with Imperfect Public Information. In: *Econometrica* 62 (1994), September, Nr. 5, S. 997–1039

[FPSS05] FEIGENBAUM, Joan ; PAPADIMITRIOU, Christos ; SAMI, Rahul ; SHENKER, Scott: A BGP-based mechanism for lowest-cost routing. In: *Distrib. Comput.* 18 (2005), Nr. 1, S. 61–72. – ISSN 0178–2770

[Fri71] FRIEDMAN, James W.: A Non-cooperative Equilibrium for Supergames. In: *Review of Economic Studies* 38 (1971), Januar, Nr. 113, S. 1–12

[FS02] FEIGENBAUM, Joan ; SHENKER, Scott: Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In: *In Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, ACM Press, 2002, S. 1–13

[FT91] FUDENBERG, Drew ; TIROLE, Jean: *Game Theory*. The MIT Press, 1991

[GC07] GANGSHU CAI, Xiuli C.: The Non-Existence of Equilibrium in Sequential Auctions when Bids are Revealed. In: *Journal of Electronic Commerce Research* 8 (2007), Nr. 2

[GTA99] GAMBARDELLA, L. M. ; TAILLARD, E. ; AGAZZI, G.: MACS-VRPTW: A Multiple Ant Colony System for Vehicle Routing Problems with Time Windows. Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale, 1999. – Forschungsbericht

[Gül02] GÜLCÜ, Ceki: *Short introduction to log4j*, March 2002

[JA05] JUN, Seung ; AHAMAD, Mustaque: Incentives in BitTorrent induce free riding. In: *P2PECON '05: Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–026–4, S. 116–121

[KC03] KEPHART, Jeffrey O. ; CHESS, David M.: The Vision of Autonomic Computing. In: *Computer* 36 (2003), Januar, Nr. 1, S. 41–50. – ISSN 0018–9162

[KL88] KAY, J. ; LAUDER, P.: A fair share scheduler. In: *Commun. ACM* 31 (1988), Nr. 1, S. 44–55. – ISSN 0001–0782

[KP97] KRISHNA, Vijay ; PERRY, Motty: Efficient Mechanism Design / EconWPA. 1997 (9703010). – Game Theory and Information

[KR02] KUROSE, James F. ; ROSS, Keith: *Computer Networking: A Top-Down Approach Featuring the Internet*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0201976994

[KS93] KLEMM, L. ; SCHWELLENBACH, D.: Leistungsmessung in verteilten Systemen: Meßmodell, Implementierung, Anwendung. (1993)

[Kwi05] KWIEK, Maksymilian: Reputation and Cooperation in the Repeated Second-price Auctions / Economics Division, School of Social Sciences, University of Southampton. 2005 (0607). – Discussion Paper Series In Economics And Econometrics

[LR99] LEWIS, L. ; RAY, P.: Service level management definition, architecture, and research challenges, 1999

[Mar08] MARINESCU, Dan: *Design and Evaluation of Self-Management Approaches for Virtual Machine-Based Environments*, FH Wiesbaden, FB Design Informatik Medien, Diplomarbeit, Februar 2008

[MCWG95] MAS-COLELL, Andreu ; WHINSTON, Michael D. ; GREEN, Jerry R.: *Microeconomic Theory*. Oxford University Press, 1995. – ISBN 0195073401

[Mil89] MILLS, D. L.: *RFC 1128: Measured performance of the Network Time Protocol in the Internet system*. Oktober 1989. – Status: UNKNOWN.

[Mil92] MILLS, David L.: *RFC 1305: Network Time Protocol (Version 3) Specification, Implementation*. März 1992. – Status: DRAFT STANDARD.

[MS83] MYERSON, Roger B. ; SATTERTHWAITE, Mark A.: Efficient mechanisms for bilateral trading. In: *Journal of Economic Theory* 29 (1983), April, Nr. 2, S. 265–281

[MVN44] MORGENSTERN, Oskar ; VON NEUMANN, John: *Theory of Games and Economic Behavior*. Princeton University Press, 1944. – ISBN 0691003629

[MWJ+07] MÜHL, Gero ; WERNER, Matthias ; JAEGER, Michael A. ; HERRMANN, Klaus ; PARZYJEGLA, Helge: On the Definitions of Self-Managing and Self-Organizing Systems. In: *KiVS 2007 Workshop: Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS 2007)*, Springer, 2007

[Nao01] NAOR, Moni: Cryptography and mechanism design. In: *TARK '01: Proceedings of the 8th conference on Theoretical aspects of rationality and knowledge*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001. – ISBN 1–55860–791–9, S. 163–167

[Nas50] NASH, John: Equilibrium Points in N-Person Games. In: *Proceedings of the National Academy of Sciences of the United States of America* 36 (1950), Nr. 48-49

[Nas51] NASH, John: Non-Cooperative Games. In: *The Annals of Mathematics* 54 (1951), September, Nr. 2, S. 286–295

[NR01] NISAN, Noam ; RONEN, Amir: Algorithmic Mechanism Design. In: *Games and Economic Behavior* 35 (2001), April, Nr. 1-2, S. 166–196

[NRTV07] NISAN, Noam ; ROUGHGARDEN, Tim ; TARDOS, Eva ; VAZIRANI, Vijay V.: *Algorithmic Game Theory*. New York, NY, USA : Cambridge University Press, 2007. – ISBN 0521872820

[Ogc01] OGC: *Service Delivery (It Infrastructure Library Series)*. Not Avail, 2001. – ISBN 0113300174

[Pap93] PAPADIMITRIOU, Christos H.: *Computational Complexity*. Addison Wesley, 1993. – ISBN 0201530821

[Par01] PARKES, David C.: *Iterative combinatorial auctions: achieving economic and computational efficiency*. Philadelphia, PA, USA, Diss., 2001. – Supervisor-Lyle H. Ungar

[Par07] PARKES, David C.: Online Mechanisms. In: NISAN, Noam (Hrsg.) ; ROUGHGARDEN, Tim (Hrsg.) ; TARDOS, Eva (Hrsg.) ; VAZIRANI, Vijay (Hrsg.): *Algorithmic Game Theory*. Cambridge University Press, 2007, Kapitel 16

[PS04] PARKES, David C. ; SHNEIDMAN, Jeffrey: Distributed Implementations of Vickrey-Clarke-Groves Mechanisms. In: *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 1–58113–864–4, S. 261–268

[RKKC05] RIDGE, Enda ; KUDENKO, Daniel ; KAZAKOV, Dimitar ; CURRY, Edward: Moving Nature-Inspired Algorithms to Parallel, Asynchronous and Decentralised Environments. In: *SOAS*, 2005, S. 35–49

[Rot07] ROTHKOPF, Michael H.: Thirteen Reasons Why the Vickrey-Clarke-Groves Process Is Not Practical. In: *OPERATIONS RESEARCH* 55 (2007), Nr. 2, S. 191–197

[RRC+06] ROLIA, Jerry ; ROLIA, Jerry ; CHERKASOVA, Ludmila ; ARLITT, Martin ; MACHIRAJU, Vijay ; MACHIRAJU, Vijay: Supporting application quality of service in shared resource pools. In: *Commun. ACM* 49 (2006), Nr. 3, S. 55–60. – ISSN 0001–0782

[San96] SANDHOLM, Tuomas: Limitations of the Vickrey auction in computational multiagent systems. In: *In Proceedings of the Second International Conference on Multiagent Systems (ICMAS-96*, AAAI Press, 1996, S. 299–306

[SK05] SCHMID, Markus ; KRÖGER, Reinhold: Selbstmanagement-Anstze im eBusiness-Umfeld. In: *PIK - Praxis der Informationsverarbeitung und Kommunikation* (2005), Nr. 28 / 4, S. 211–216

[SK08] SCHMID, Markus ; KRÖGER, Reinhold: Decentralised QoS-Management in Service Oriented Architectures. In: MEIER, René (Hrsg.) ; TERZIS, Sotirios (Hrsg.): *DAIS* Bd. 5053, 2008. – ISBN 978–3–540–68639–2, S. 44–57

[SP03] SHNEIDMAN, Jeffrey ; PARKES, David C.: Rationality and Self-Interest in Peer to Peer Networks. In: *2nd Int. Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003

[Ste08] STEIMLE, Jürgen: *Algorithmic Mechanism Design*. Springer-Verlag Berlin Heidelberg, 2008

[Tan01] TANENBAUM, Andrew S.: *Modern Operating Systems*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2001. – ISBN 0130313580

[TBMM04] THOMPSON, Henry S. (Hrsg.) ; BEECH, David (Hrsg.) ; MALONEY, Murray (Hrsg.) ; MENDELSOHN, Noah (Hrsg.): *XML Schema: Part 1: Structures*. Second. W3C, 2004 (W3C Recommendation)

[TTU06] TRUMLER, Wolfgang ; THIEMANN, Tobias ; UNGERER, Theo: An Artificial Hormone System for Self-organization of Networked Nodes. In: *Proceedings of the 1st IFIP International Conference on Biologically Inspired Cooperative Computing (BICC 2006)* Bd. 216, Springer, August 2006. – ISBN 978–0–387–34632–8, S. 85–94

[Vas07] VASILIEV, Yuli: *SOA and WS-BPEL*. Packt Publishing, 2007. – ISBN 184719270X

[Vic61] VICKREY, William: Counterspeculation, Auctions, and Competitive Sealed Tenders. In: *The Journal of Finance* 16 (1961), Nr. 1, S. 8–37

[YBT05] YU, Jia ; BUYYA, Rajkumar ; THAM, Chen-Khong: QoS-based Scheduling of Workflow Applications on Service Grids. Melbourne, Australia : Grid Computing and Distributed Systems Laboratory, University of Melbourne, 2005. – Forschungsbericht

[ZQS05] ZOU, Deqing ; QIANG, Weizhong ; SHI, Xuanhua: A Formal General Framework and Service Access Model for Service Grid. In: *ICECCS '05:*

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2284–X, S. 349–356