Verteilte Prozesskontrolle in ressourcenbasierten Architekturen

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Martin Sugioarto

Dortmund

2013

Tag der mündlichen Prüfung:	28. Mai 2013
Dekan:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter:	Prof. Dr. Jakob Rehof
	Prof. Dr. Bernhard Steffen

Ich möchte mich insbesondere bei Prof. Dr. Jakob Rehof herzlich bedanken für das Vertrauen, die Betreuung und die Unterstützung während dieser Arbeit.

Einen besonderen Dank möchte ich auch Prof. Dr. Bernhard Steffen aussprechen, der mich als Zweitgutachter begleitet hat.

Weiterhin danke ich Prof. Dr. Heinrich Müller für die Leitung des Vorsitzes in der Prüfungskommission. Auch möchte ich Dr. Stefan Dißmann als weiteres Mitglied der Prüfungskommission danken.

Außerdem möchte ich meinen Dank an die Mitarbeiter des Lehrstuhls XIV der Fakultät Informatik und des Fraunhofer ISST in Dortmund richten, deren Kommentare und Anregungen mir sehr geholfen haben.

Schließlich danke ich meiner geliebten Frau Dewy und meinem frisch auf die Welt gekommenen Sohn Evan für all die Freuden, die sie mir begleitend bereitet haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Vorwort

Parallele Systeme und verteilte Architekturen sind kein neues Thema. Wir haben zunehmend damit zu tun, die Systeme so zu erstellen und einzurichten, dass die parallelisierte Arbeitsweise zur Effizienz beiträgt. Im Bereich der Geschäftsprozesse und ihrem Management sind bei diesem Vorhaben komplexe Architekturen entstanden, die auf Frameworks basieren, die wiederum vom Charakter komplex sind. Als eine Art Kompromiss auf diesem Weg sind Workflow-basierte Systeme entstanden, die dem Entwickler die Einfachheit in der Konzeption größerer Zusammenhänge geben („programming in the large“ [DK75]). Geschäftsprozesse sind in den Fokus der Wirtschaft geraten, weil sie einfach zu verstehen geworden sind.

Um das Thema Geschäftsprozesse hat sich ein lebendiges Feld entwickelt, welches von der Wissenschaft und der Wirtschaft vorangetrieben wird. In den Anfängen der Prozesssteuerung (auch „Prozessmanagement“) haben Unix-Systeme mit ihrer flexiblen Arbeitsweise durch die übergeordnete Schicht, die Skriptausführungen erlaubt hat, sehr schön innovativ gezeigt, dass die Steuerung noch eine Ebene hat, die über einzelne Anwendungsausführung hinaus funktioniert. Die einfache Konzeption des Kommandozeilenaufrufs und des Mechanismus der Ein- und Ausgabeströmen hat es erlaubt, Systeme zu entwickeln, die aus vielen Einzelkomponenten bestehen und zusammen eine Gesamtfunktionalität anbieten. Diese Systeme funktionieren dermaßen zufriedenstellend, dass viele Firmen bis heute auf diese Weise prozessgestützt arbeiten.

Es ist nicht zu unterschätzen, dass die Einfachheit des Systems hier entscheidend zum Erfolg geführt hat. Es wurden dienstebasierte Systeme aufgebaut, die lokal und durch das Netzwerk erreicht werden konnten und der Begriff der *dienstebasierte Software-Architektur* („service-oriented architecture“ [Erl07]) rückte in den Fokus der Entwicklung. Ein System, welches einfache Parameterübergaben erlaubt und Prinzipien wie Dateien, Datenströme und Netzwerkkommunikation unterstützt, ist einfach zu begreifen und wird deswegen oft gegenüber komplexen und vielmals spezialisierten Lösungen favorisiert.

In der letzten Zeit wurde viel daran getan, um Unix-artige Systeme zu entwickeln, die nicht proprietär sind. Dies ist eine weitere Chance für Unternehmen,

ihre Geschäftsprozesse in diesem Umfeld noch eine lange Zeit am Leben zu erhalten. Die Vielfalt der Lösungen aus der Open-Source Anwendungsentwicklung, die auf diese Art von Systemen konzipiert sind, gibt eine enorme Energie, um diesen Status Quo der Systementwicklung und Architekturen noch lange voranzutreiben. Die Einfachheit dieser Architekturen ist auf Entwickler ausgerichtet. Diese Art von Architektur wurde von Entwicklern für Entwickler geschaffen und deswegen wird diese auch sehr gut verstanden und größtenteils im Konsens mit dem öffentlichen Interesse entwickelt.

Seit nicht all zu langer Zeit hat sich das Denken über Geschäftsprozesse gewandelt. Das Verkaufsargument ist die modellgestützte Entwicklung, die einen Anwender ansprechen soll, der von tiefergehenden professionellen Entwicklung nichts verstehen müsse, um Geschäftsprozesse zu entwickeln. Die Ursprungsidee, die Prozesswelt weitgehend zu formalisieren und Werkzeuge zu entwickeln, die die überliegenden steuernden Sprachen für Entwickler interessant zu gestalten, ist in den Hintergrund gerückt. Alles was zählt, ist die Einfachheit der Bedienung und die daraus resultierende Einfachheit der Lösungskonzepte, was zu vereinfachten Entwicklungswerkzeugen führt. Solche Werkzeuge vertreiben auch den erfahrenen Entwickler, der nach neuen, mächtigen Konzepten sucht, um Aufgaben effizient zu lösen.

Das Thema Geschäftsprozesse kam für mich zum ersten Mal im Jahr 2005 zur Sprache, an der (damals noch[1]) Universität Dortmund am Lehrstuhl 5 für Programmiersysteme in Zusammenarbeit mit Prof. Dr. Bernhard Steffen. Sein Mitarbeiter und Doktorand, Ralf Nagel, hat mir Gelegenheit gegeben, mich in sein Dissertationsthema und das resultierende jABC-Framework [Nag09] einzuarbeiten. jABC vereinte das Modell und die Ausführung, um leicht verständlich und nachvollziehbar, Geschäftsprozesse auszuführen und zu analysieren. Später wurde die Ausführung stärker modular betrachtet und zudem parallelisierte Ausführung ermöglicht. Es hat sich auch als ein Vorteil erwiesen, das Modell von der Ausführung zu trennen, weil produktive Lösungen auf Servern oft ohne grafische Ausgaben arbeiten. jABC hat mich ebenfalls dazu bewegt, mir erste Gedanken darüber zu machen, wie mit Parallelismus in Geschäftsprozessen gearbeitet wird. Das Ein- und Ausgabeverhalten der Geschäftsprozesskomponenten (in jABC kurz „SIB“ genannt), gab mir auch den ersten Hinweis, sich mit datengesteuerten Prozessen zu beschäftigen.

Bei der Entwicklung kommunizierender verteilter Systeme, habe ich nach Werkzeugen gesucht, die vor allem mit Hilfe von Daten gesteuert werden, die aktuell in einem System vorgehalten werden. Mich begeisterte dabei die Möglichkeit, Prozesse datenbasiert zu unterstützen, die zusammenhängend größere Aufga-

[1] heute: Technische Universität Dortmund

ben erledigen. Vordergründig habe ich dabei Systeme betrachtet und analysiert, bei denen die Trennung von Laufzeit und Daten eine wichtige Rolle spielt.

Die attraktivste Architektur für meine Experimente auf dem Gebiet der Prozesskontrolle in datenbasierten Systemen ist auch eines der besten und erfolgreichsten verteilten Architekturen, das *World Wide Web* . Warum diese Architektur so groß und erfolgreich wurde, erklärt die Dissertation von Roy T. Fielding, die den Begriff „Representational State Transfer“ [Fie00] im Bereich der Web-Dienste eingeführt hat. Die guten Ideen, die in dieser großartigen Arbeit zusammengefasst wurden, werden leider heutzutage nicht alle umgesetzt. Aber man kann auch feststellen, dass wo die grundlegenden Prämissen dieser Architektur verletzt werden, die Systeme und Anwendungen aus diversen Gründen qualitativ schlecht werden. Das schlägt sich insbesondere darin nieder, dass ein solches fehlentwickeltes System ein Problem mit Skalierung und Verteilung bekommt, obwohl eigentlich alle Mittel zu diesem Zweck bereit vorliegen. Ein sehr bekanntes Beispiel, welches oft im Zusammenhang mit der verteilten Architektur des Webs kritisiert wird, ist SOAP [GHM+07].

Zusammenfassung

Diese Arbeit befasst sich mit verteilten Software-Systemen und -Architekturen, die, wie das World Wide Web, datengetrieben arbeiten. Eine bekannte Ausprägung dieser Architekturen beschreibt Roy Thomas Fielding als den Architekturstil namens REST [Fie00]. Der Fokus liegt hier auf der Verbesserung der Parallelisierung moderner Systeme durch den Einsatz von Techniken bei der Umsetzung von solchen Systemen, die nicht nur auf der Betreiberseite skalierbar sind, sondern auch mit den Nutzern gut skalierbar gestaltet sind. Dies erfordert jedoch, dass man Skalierbarkeit auf Nutzerseite für sich nutzbar macht.

Der Begriff der *Prozesskontrolle* wird in dieser Arbeit gründlich untersucht, um die Bearbeitung von verteilten Aufgaben zu unterstützen und insbesondere um die kooperative Arbeit an verteilten Geschäftsprozessen zu erlauben. Ein spezielles Beispiel, mit dem sich hier beschäftigt wird, ist das Thema *Crowdsourcing*. Anwendungen, die mit Crowdsourcing betrieben werden benutzen die Stärken der Architektur des World Wide Webs, in welcher die Arbeitskraft verteilt und entkoppelt (also unbeeinflusst) zur Verfügung steht (oder eben gerade *nicht* verfügbar ist). Die Herausforderung dabei ist, mit den Daten und der enormen Rechenkraft in der Architektur geeignet umzugehen, damit eine große Aufgabe oder ein Prozessverlauf organisiert werden kann. Im *World Wide Web*, das hier in dieser Arbeit stets als ein gutes Beispiel zur Demonstration der Überlegungen nehme, stehen dazu nicht viele Mittel zur Verfügung, weil das verteilte Verhalten des Gesamtsystems im Detail unvorhersagbar und chaotisch ist. Um einen Geschäftsprozess jedoch zu unterstützen, muss Ordnung in das parallelisierte Geschehen gebracht werden und diese Ordnung entsteht durch die Gestaltung der Kooperation von komponierten Systemen.

Nach einer kurzen Analyse der herkömmlichen, bekannten Prozessmodelle wird schlussgefolgert, dass diese Modelle nicht geeignet sind, um datenbasiertes Rechnen zu unterstützen, welches sich die zur Verfügung stehende verteilte Rechenkraft zu nutze macht. Um auf den richtigen Ansatz zu kommen, muss man die Entwicklungen der letzten Jahre auf dem Gebiet der Prozesse noch einmal revidieren und das zentralisierte Geschäftsprozessmanagement mit Hilfe von Orchestrierung vermeiden.

Als Schlussfolgerung ergibt sich ein Prozessmodell, welches zunächst datenbasiert arbeitet und sich an die Ideen von Fieldings Architekturstil REST anlehnt. In Fieldings Arbeit wurde REST sehr allgemein aufgefasst. Es wurden die Entstehung, die Ideen und Gründe genannt, den Architekturstil auf diese Weise entworfen zu haben. Im Laufe der Zeit ergaben sich viele Diskussionen in der öffentlichen Entwicklergemeinde, welche Bedeutung REST hat. Der größte Teil der Entwickler ist sich einig, dass REST ein passenderer Architekturstil für Web-Dienste ist als zum Beispiel SOAP [GHM+07]. Des Weiteren weiß man mit *REST als Programmiermodell* nicht viel anzufangen, weil dies durch Fielding Arbeit nicht abgedeckt worden ist. In dieser Arbeit wird REST im Hinblick auf die Fähigkeiten zur Prozesssteuerung untersucht. Dazu wird das Programmiermodell konkretisiert und durch die ausführende Schicht der *Agenten* erweitert.

Um die Architekturen und ihre Eigenschaften als Programmiermodell besser verstehen und darüber argumentieren zu können, wird der Architekturstil zum ersten Mal systematisiert betrachtet und eine semiformale Beschreibung eingeführt. Darin werden die Begriffe des Architekturstils präzisiert, um sie später konkreter auf HTTP [F+99] anwenden zu können, welches das Kommunikationsprotokoll im *World Wide Web* und eine Variante von REST ist. Das weiterführende Ziel bei der Erstellung des Formalismus ist, die Aspekte der Verteilung, Redundanz und Entkopplung zu bewahren, worin Besonderheit dieses Ansatzes liegt.

Als Resultat dieser Arbeit werden genauere Einsichten gegeben, welche Eigenschaften die Architekturkomponenten und die Protokolle in entkoppelten, verteilten Architekturen (die nach den Prinzipien von REST gebaut worden sind) haben sollten, damit Möglichkeiten zur Prozesssteuerung, wie man sie aus der Welt der Geschäftsprozesse kennt entstehen. Das ganze natürlich basierend auf einer „Welt“, die vor allem aus Daten besteht und den aktiven Agenten, die diese Datenwelt wandeln und transformieren. Die Herausforderung hierbei war, diese Welt im Standardfall *parallelisiert* zu betrachten und sie erst zu *synchronisieren*, wenn es der Prozessverlauf erforderlich macht. Dies ist eine Paradigmenumkehr, die im Kontrast zu orchestierten Geschäftsprozessen steht, in denen die Operation der Sequenz dominierend ist.

Ausgehend von der Definition der Ressourcen- und Agentenwelt wird festgestellt, dass es in diesem entstandenen Architekturstil durchaus Mittel gibt, die Prozesssteuerung unter gewissen Voraussetzungen erlauben. Insbesondere ist das monotone Verhalten, einerseits in Ressourcen und andererseits in den Agentenprotokollen, ein wichtiges Mittel zur Synchronisierung von Ressourcenzuständen und um die Ausführung von verteilten Prozessen unterstützen zu können. Anhand der später vorgestellten Warteschlangentypen lassen sich Komponenten bilden, die die Monotonie dazu nutzen, um Verteilung und redundan-

te Berechnungen abstrakt behandeln zu können. Als Gegenstück in der Agentenwelt fungieren die monotonen Sequenzen, die es erlauben, Ergebnisse in der Ressourcenwelt konsistent zu halten, um Abläufe aneinander kompositional zu koppeln. Diese herausgearbeitete Wechselwirkung der Ressourcen- und Agenteneigenschaften wird am Ende der Arbeit einerseits anhand von Simulationen von Workflow- und Petri-Netzsystemen, als ein Nachweis über Äquivalenz der Paradigmen, und andererseits anhand von Beispielen demonstriert, die Szenarien für verteilte kooperative Prozesse darstellen.

Neben der theoretischen Ausarbeitung entstand zu Experimentalzwecken ein Framework namens *cgi2c*, um die resultierenden Ideen und Architektureigenschaften praktisch zu testen. Mit dieser Middleware-Komponente war es möglich REST-konforme Web-Dienste als, die in dieser Arbeit motivierten, *datenstrukturgetriebenen Ressourcen* wiederverwendbar zu entwickeln. Die Umsetzung der entsprechenden Testszenarien konnte mit Hilfe dieser Wiederverwendbarkeit des mehrfach benötigten Ressourcenverhaltens einfacher durchgeführt und anschließend gründlich untersucht werden.

Kapitel 1

Die Welt der Prozesse

Ein Prozess ist ein Konzept zur Verwaltung der Ausführung von Programmen und stammt vor allem aus dem Bereich der Betriebssysteme [Tan07]. Das Management der Prozesse hat sich gewandelt und vor allem automatisiert. Wo früher sogenannte „Operators“ für die Laufzeitzuteilung und die Steuerung der verfügbaren Systemressourcen zuständig waren, läuft heutzutage der meiste Teil automatisch ab, indem moderne Betriebssysteme Scheduling anbieten.

Heute spricht man von Prozessen in IT-Umgebungen oft im Kontext von Geschäftsprozessen und deren Automatisierung [Oul95] [Law97]. Inzwischen haben sich einige bekannte Standards zur Prozessausführung, wie *WS-BPEL* [OAS07] oder XPDL [Wor08] und zur Modellierung, wie BPMN [Obj11] und ARIS [Sch99] entwickelt und durchgesetzt. Im universitären Bereich sind ebenfalls Werkzeuge entstanden, die Modellierung und Ausführung in Form eines Frameworks realisieren. Hier sind *jABC* [Nag09] und *YAWL* [vdAH05] einige Beispiele.

Dieses Kapitel definiert einige hier in der Arbeit benutzten Begriffe und gibt einen Rückblick auf die Welt der Geschäftsprozesse und analysiert die Entwicklungen, die in den Bereichen dort entstanden sind. Es wird zusammengefasst was Prozesse sind, wie sie formuliert werden und wo Defizite herrschen im Hinblick auf Verteilung. Die resultierenden Probleme sollen als Motivation für eine andere Art von Handhabung der Prozesskontrolle verstanden werden.

1.1 Einführende Begriffsdefinitionen

Zur Einführung sind einige Grundbegriffe notwendig, um das Verständnis für die Terminologie zu bekommen, die in dieser Arbeit gebraucht werden. Weitere fachliche Begriffe, die auf diesen Grundbegriffen aufbauen, werden im weiteren Textverlauf vorgestellt und detailliert beschrieben.

Diese Grundbegriffe entsprechen nicht einer Definition, die man in der Literatur findet, sondern sind absichtlich aus der Perspektive dieser Arbeit zu sehen. Sie mögen vielleicht dem allgemeinen Verständnis entsprechen, aber sie sind auch sehr vorsichtig und so weit gefasst, und zwar so, dass man die verschiedenen Beschreibungsformen der Prozesskontrolle in den nachfolgenden Teilen, anhand dieser Begriffe, verständlich erklären kann.

Architektur ist ein Gebilde welche eine geordnete Gesamtheit an Komponenten und Aktoren in einem informationstechnischen Raum repräsentiert.

Architekturstil ist ein Prinzip, nachdem eine bestimmte Architektur aufgebaut wurde, um bestimmte Ziele und Paradigmen zu erreichen.

System ist ein in einer Architektur implementiertes funktionales Gebilde, welches eine Arbeit verrichtet, welche einen bestimmten, durch den Systementwickler vorgegebenen Zweck verfolgt.

Laufzeit beschreibt eine Zeitspanne, in der jegliche Art von Datenverarbeitung in einer Architektur passiert.

Tätigkeit/Aktion/Aktivität ist eine abstrakte Formulierung für eine Einheit, die Laufzeit braucht, um in einem oder mehreren Systemen eine Auswirkung auf die Datenverarbeitung hat.

Dienst ist eine wiederverwendbare Komponente eines Systems, die parametrisiert werden kann, um eine Aktion im System zu verursachen.

Kontrolle/Steuerung ist eine Bezeichnung für den Teil der Veränderungen in einer Architektur, der ebenfalls Laufzeit braucht, jedoch nicht in konkrete Aktivitäten einfließt, sondern deren Laufzeitzuweisung.

Prozess ist eine strukturierte Formulierung der Kontrolle, die Tätigkeiten organisiert, um eine Gesamtwirkung in einem oder mehreren Systemen zu erzielen.

Prozessmanagement ist eine über den Prozessen liegende Steuerungsschicht, die Aktivitäten, nach den im Prozess formulierten Vorgaben, dazu befähigt Laufzeit in Anspruch zu nehmen.

1.2 Beschreibungsformen der Prozesskontrolle

Wird uns Menschen eine Auswahl von Diensten oder Aktivitäten zur Verfügung gestellt, dann können wir, von der Intuition her oft sagen, um welche Art von

Unternehmen es sich handelt und nach gewisser Bedenkzeit auch, welche sinnvolle Prozesse wir damit durchführen könnten. Natürlich erfordert es eine menschenverständliche Beschreibung der jeweiligen Dienste, um zu wissen was semantisch hinter diesem passiert. Aus dem Eindruck, dass alleine das Angebot an einzelnen Diensten, vieles an der Art und Weise wie der Prozess aussieht impliziert, folgte die Entscheidung, die Möglichkeiten auf Prozessmanagementebene zu untersuchen und eine andere Sicht auf Prozesse zu motivieren.

Die Intuition, dass eine globale Prozessbeschreibung eigentlich unnötig ist, wenn klar ist, welche Dienste und Aktivitäten zur Verfügung stehen, entstammt aus den Überlegungen bezüglich des Aspekts der Kontrolle. Diese beschreibt Ordnungen auf gewissen Abfolgen der Dienstnutzung. Hier lassen sich viele Fragen stellen. In welcher Form kann man die Kontrolle vom Diensteangebot „abtrennen“ und vielleicht aus der Prozessbeschreibung „herausfiltern“, sodass Prozessmanagement unnötig wird? Wenn die Kontrolle nicht formuliert werden muss, dann sind Prozesse unmissverständlich einfach. Gibt es solche Prozesse? Welche Möglichkeiten gibt man auf, wenn man Prozesse mit dermaßen trivialer Ausführungssteuerung entwickelt?

Diese Frage stellte sich als sehr schwierig heraus, weil sie zu weit in eine unbekannte Welt vorausgreift. Es fängt schon damit an, dass das „Herausfiltern“ der Kontrolle nicht einfach ist. Es stellten sich außerdem Fragen, wie eine Architektur aussieht, wo Kontrolle und Ausführung anders gehandhabt werden. Was für ein System ergibt sich, wenn man auf Kontrolle verzichtet und was ist wenn die Kontrolle dynamisch ist und nicht in den Händen einer zentralisierten Komponente für das Prozessmanagement?

Die Abwesenheit von Laufzeit in einer Architektur machte das Problem spannend. Eine Architektur, die hauptsächlich aus einer Datenwelt besteht, also datenorientiert arbeitet, eröffnet eine Möglichkeit, diese als „stabil“ anzusehen.

Bevor man jedoch zu überstürzt die ganze Welt der Geschäftsprozesse umwirft und nach neuen Paradigmen sucht, sollte man sich bestehende Verfahren anschauen und Inspiration davon schöpfen, was gut an diesen ist.

1.2.1 Workflows

Eine der bekanntesten Formen in der Geschäftsprozessentwicklung sind *Workflows* [vdAtHKB03]. Ein Workflow ist ein Netz, das eine Folge von Aktivitäten und den zeitlichen Zusammenhang beschreibt, wie diese Aktivitäten aufzurufen sind. Ein Workflow ist mehrfach instanziierbar und unterstützt die parallele Ausführung in zwei verschiedenen Arten. Die erste ist die explizit formulierte Parallelisierung innerhalb des Workflows, die sich auf eine Instanz, also auf den

resultierenden Prozess auswirkt. Die andere ist die Unterstützung von Workflows für mehrere Prozessinstanzen, die durch das Management getrennt werden. Prozesse haben im allgemeinen ein zentrales Prozessmanagement, das die Steuerung übernimmt.

Workflows haben das Problem, dass sie nicht mit einer frei verteilten Umgebung zusammen funktionieren. Das Management verlässt sich auf eine stabile Welt drumherum, die keinerlei Änderungen unterliegt. Eine entkoppelte Änderung kann einen Prozess ungültig machen, wie es später im Teil 3.9 verdeutlicht wird. Die echte Welt funktioniert entkoppelt und parallel. Diese Eigenschaften werden in Workflows völlig vernachlässigt, zu Gunsten der Einfachheit in der Modellierung. Würde ein Workflow-Entwickler versuchen, die Funktionsweise einer verteilten Architektur, wie des Webs, zu formulieren, würde er vor einem Problem stehen, das schlecht handzuhaben ist. Das Web hat weder einen Startzustand, noch einen Endzustand und keine übersichtliche Darstellung der in dieser Architektur gerade ablaufenden Aktivitäten. Die Ausführung in der Architektur ist vor allem ungeordnet und wird bei Bedarf erst durch Synchronisierung in eine Ordnung gebracht.

1.2.2 Semantische Prozessmodellierung

Ein weiterer Schritt in Richtung des automatischen Prozessmanagements ist der Einsatz von Inferenz auf der semantischen oder logischen Beschreibung der einzelnen Dienste. Diese Beschreibung dient später dazu, benötigte Dienste während der Ausführung an der Kompatibilität zum aktuell ablaufenden Prozess zu erkennen. Anstatt eine kompatible Schnittstelle zu suchen, für die eindeutig der Prozess geschrieben worden ist, wird ein semantisches Matching durchgeführt nach einem Dienst, der eine Aufgabenstellung verstehen würde. Mit Hilfe einer semantischen Beschreibung lassen sich die feineren Inkompatibilitäten während der Anbindung des Dienstes beheben. Die semantische Beschreibung umschreibt den Dienst von seiner Funktionsseite und nicht anhand von konkreter Diensteschnittstelle und erlaubt damit eine vielfältigere Nutzung in anderen Kontexten. Ein mögliches Vorgehen bei dieser Art von Synthese wurde im Papier über CAD-METAFrame [MS96] erläutert, in dem auch die „Semantic Linear-Time Temporal Logic“ als die Grundlage der Inferenz gewählt wurde.

Obwohl Mittel zur semantischen Beschreibung mittels OWL-S [MBH+04] oder WSDL-S [AFM+05] schon lange in einer standardisierten Form existieren, ist die Entwicklung in diesen Bereichen zum Stillstand gekommen. Ein Grund dafür mag sein, dass die logische Beschreibung für Entwickler zu komplex und nicht ergiebig genug ist, um Vorteile daraus zu bekommen. Ein anderer Grund ist, dass die Werkzeugunterstützung sehr dürftig ist und bei genauer Untersuchung

stellte sich heraus, dass die Werkzeuge sogar auf die Beispiele hin entwickelt und optimiert wurden, was keine generelle Lösung darstellt.

Die Einfachheit und Reife der Alternativen aus dem Bereich der expliziten Modellierung und der Übersetzung in die herkömmliche statische WS-BPEL [OAS07]-Sprache oder anderen Workflow-orientierten Modellierungsarten ist viel attraktiver. Die Essenz aus dieser semantischen Bewegung ist, dass man Sachverhalte, die im Kern einfach sind, nicht unnötig verkomplizieren sollte. Dann haben sie erst eine Chance ein langlebiger Standard zu werden.

1.2.3 Abhängigkeitsmodelle

Ein anderer Ansatz, Prozessabläufe zu steuern, ist, diese durch Restriktionen deklarativ zu beschreiben. Wenn man Ein- und Ausgabeparameter umschreibt und diesen Typen zuweist, ergeben sich teilweise zwangsläufige Ablaufreihenfolgen. Dieses Verfahren basiert auf der Bildung von Abhängigkeiten. Diese können auch explizit formuliert werden und sind ein einfaches Prinzip, um einen Ablauf zu konstruieren, welcher sogar parallele Ausführung ermöglicht. Was dabei sehr interessant ist, ist die Tatsache, dass die Parallelisierung von Aufgaben durch Unterspezifikation geschieht. Das heißt, dass wenn man, im Gegensatz zu herkömmlichen Workflow-Sprachen, die ineffiziente Alternative der Sequenz modellieren möchte, muss man implizit oder explizit Synchronisierung verlangen. Ein Benutzer wird hiermit mit einem einfachen Modell dazu hingeleitet, Parallelisierung auszunutzen. Diese Idee bringt eine wünschenswerte Eigenschaft, die wir in der Arbeit angehen und erweitern werden.

Eine formale Basis zur Feststellung wie die sequentielle Komposition von Aktivitäten in parallele Ausführung unter gewissen Kriterien umgestaltet werden kann, bietet das Papier „Weak Sequential Composition“ [RW94]. Dieser Ansatz verbindet die Workflow-orientierte Modellierungsart mit der der Abhängigkeiten. Es greift das bereits angesprochene Problem auf, dass Workflows die Sequenzialisierung als Standardmechanismus verwenden und damit die Entwickler dazu verleiten, das Potenzial zur Parallelisierung einfach zu übersehen.

1.2.4 Multiagentensysteme

Eine Lösung, die in die richtige Richtung bezüglich der parallelisierten Abarbeitung von Problemen einschlägt, sind die Multiagentensysteme [SLB09]. Agenten sind ein hervorragender Ansatz, um verschiedene Aspekte zu erfüllen, die für verteiltes Rechnen erstrebenswert sind. Es sind Ausfallsicherheit und Entkopplung bezüglich der Welt, auf der sie agieren. Wie sich aber schnell zeigt, sind

Multiagentensysteme ganz anders aufgestellt, als man es sich in verteilten Architekturen wünscht. In Architekturen aus Computer-Netzen, welche für unser Verständnis Client-Server-Architekturen sind, ist es zwar möglich, eine Kommunikationsarchitektur aufzubauen, die den Agenten erlaubt miteinander zu kommunizieren, aber diese würde nur innerhalb eines Systems sinnvoll sein. Die einzelnen Systeme in einer Architektur sind meistens heterogene Anwendungen. Sie bieten Dienste an, die einmalig sind und Multiagentensysteme zeigen ihre Stärken gerade im homogenen Architekturen, wo Redundanz und paralleles Rechnen miteinander kombiniert werden.

Multiagentensysteme verwenden außerdem keine Trennung von Daten und Laufzeit und sind auf einander im System angewiesen. Nur im Verbund ergibt sich hier eine verteilte Funktionalität, die aus Daten und Laufzeit besteht innerhalb eines jeden einzelnen Agenten. Das lokale Verhalten von Agenten ist für eine festgelegte Anwendung stets gleich, denn darauf basiert das Prinzip der Redundanz und Austauschbarkeit. Die Agenten funktionieren vom Charakter wie Peer-to-Peer-Netze [Kou03] mit gleichförmigen Knoten, die in sich jeweils die gleiche Funktionalität lokal kapseln. Um Anwendungsvielfalt zu unterstützen, empfiehlt es sich jedoch, die Agentenwelt mit einer Kombination aus Individuen, für entkoppelte, und gleichförmigen Gruppen, für verteilte und redundante Arbeit, auszustatten.

Man sieht daher leicht, dass Multiagentensysteme völlig andere Architekturmerkmale unterstützen, als man es sich für Prozessarchitekturen und den darauf laufenden Prozessen wünscht. Eine Eigenschaft, die Agenten attraktiv macht, ist das Konzept der Benutzung von Sensorik, die auch in entsprechender Form bei Menschen vorhanden ist. Dieses Merkmal wird später noch einmal aufgegriffen, wenn es darum geht, die aktuelle Situation zu erkennen, um Aufgaben zu identifizieren.

1.2.5 Complex Event Processing

Ereignisse oder Nachrichten sind ein asynchroner Kommunikationsmechanismus zur Ansteuerung von einzelnen Aktivitäten eines Prozesses. *Complex Event Processing* (CEP) [Luc05] behandelt die Abarbeitung von Prozessen anhand von den Ereignissen, die in den Systemen passieren. Die dazugehörige Managementkomponente ist im Allgemeinen ein geschlossenes System, innerhalb dessen mit Filterung, Abhängigkeiten und logischen Verknüpfungen global auf das Auftauchen einer Nachricht gewartet wird, die eine Abfolge von Aktionen anstößt.

Diese Architekturstil basiert auf Aktionen und Reaktionen und ihren Rückkopplungen. Ein solches nachrichtenbasiertes System kann sehr schnell unübersicht-

lich werden, obwohl die Architekturen eher zentralisiert sind, durch die nachrichtenverarbeitende Middleware. Wie auch in den verteilten Architekturen, ist es hier schwierig einen festen Zustand zu erfassen, jedoch liegt es nicht daran, dass der Zustand verteilt und entkoppelt ist und das System zu groß ist, sondern daran, dass Nachrichten in verschiedenen Verarbeitungszuständen sind und diese Situation eine Mischung aus Laufzeitzustand und Datenzustand ist. Möchte man die Managementkomponente temporär ausschalten, ist das Problem der späteren Restaurierung des Laufzeitszustands nicht unerheblich.

Interessant ist, dass CEP durch den asynchronen Mechanismus Systeme auf interessante Weise anbinden kann. Man braucht zwar Nachrichtenregeln, die durch die Managementkomponente ausgewertet werden, aber eine Anbindung eines neuen Systems ist zu jeder Zeit möglich, falls das System keine Nachrichten aus der Vergangenheit braucht.

1.2.6 Datenorientiertes Modell

Eine andere Art von Prozessorganisation bieten dokumenten- und datenorientierte Abläufe. Hier liegt der zentrale Punkt auf der Verarbeitung von festgelegten Entitäten. Das Interessante an Datenorientierung ist, dass die Daten im System den Zustand in der Welt reflektieren. Im Gegensatz zu Agentensystemen ist es hier möglich, Daten explizit persistent zu halten, was bei Computer-Netzen keine schwierige Aufgabe ist. Aber der Nachteil hierbei ist, dass die Verwaltung parallelisierter Funktionalität statisch ist. Und dies ist wiederum ein Nachteil verglichen mit Agenten, die die Verteilung dynamisch regeln. Es gibt mehrere Verfahren wie datenorientiert gearbeitet werden kann. Es wird aber größtenteils angenommen, dass Ressourcen ihren Ort wechseln und zur verarbeitenden Komponente, die einen festen Standort hat, gereicht werden. Dieses System erfordert eine darüberliegende Prozesssteuerungskomponente, die vordefinierte Geschäftsprozesse ablaufen lässt. Ein manueller Eingriff in das System lässt sich sicherlich einfacher realisieren als bei Workflow-Management-Systemen, weil die Datensteuerung ein gutes Verfahren ist, um transparente Systeme zu erhalten, die die Möglichkeit anbieten, einen Prozess, bei Ausfall des Systems, wieder schnell ohne Hilfskonstrukte ans Laufen zu bekommen. Dazu muss man die persistenten Entitäten nur mit dem Status versorgen, welcher den den Stand der Verarbeitung festhält.

Wie bei herkömmlichen Prozesssystemen, wird bei dokumentenbasierten Systemen ein globaler Prozess entwickelt, welcher die Dokumentenverarbeitung vorantreibt. Die Einzelkomponenten werden hierzu zusammenverbunden und mit Hilfe von Spooling- oder Pipeline-ähnlichen Mechanismen verarbeitet. Die Standorte wo die Entitäten gespeichert werden, sind zumeist einfache Datenspei-

cher, die keinerlei Intelligenz anbieten, außer Daten parat zu halten. Die ganze Rechenkraft befindet sich zwischen diesen Datenspeichern und verarbeitet die Entitäten in Etappen. Dabei ist die Steuerung in solchen Prozessen nicht dezentral. In meisten Fällen ist das Management eine einzige Komponente, die über dem Datenfluss wacht und ihn steuert. Um in einem datenorientiertem Prozesssystem Ausfallsicherheit zu erhöhen, gilt es, das Prozessmanagement mit Mitteln auszustatten, die die Entitäten verteilt bearbeiten können und diese Verteilung mit Hilfe von Load-Balancing-Algorithmen unterstützen. Die Verteilung selbst wird anhand von vervielfachten, verarbeitenden Komponenten realisiert.

Datenbasierte Prozesse sind etwas Interessantes, womit man gut anfangen kann eine Architektur aufzubauen, die aber einer Erweiterung bedarf, um entkoppelte und verteilte Funktionalität anzubieten.

1.3 Entwurfsprinzipien für Prozessarchitekturen

Bei der Betrachtung von allen Modellen sind noch mehr Eigenschaften aufgefallen, die wünschenswert wären. In diesem Teil wird der Nutzen analysiert und motiviert.

1.3.1 Integration von Menschen

Alle bisher erwähnten Prozessmanagementsysteme kränkeln an einer wichtigen Herausforderung. Sie bieten keine Lösungen an, mit dem „Nichtdeterminismus“ fertig zu werden, den der Mensch darstellt. Dabei ist der Mensch selbst oft die letzte Instanz, die beurteilen kann, wie eine Arbeit erledigt werden soll. Läuft etwas falsch, kann es oft nur ein Mensch beurteilen. Die Einbindung von Menschen als Aktoren, die mit Rechenkraft ausgestatten sind, sollte in verteilten Systemen nicht unterschätzt werden. Menschen können vielfältigere Aufgaben, die für Maschinen nicht oder schwierig berechenbar sind, sehr effizient lösen.

Erfordert eine einzelne Aufgabe spezielle Zuwendung, möchte man ungerne diese einmalige Abweichung für den Prozess ausformulieren. In einem Workflow-basierten Prozess verletzt man mit einer Änderung die Konsistenz zwischen Daten und laufendem Prozess. Im allgemeinen Fall ist eine Abbildung von einem alten Prozesszustand auf einen geänderten neuen nicht automatisch berechenbar. Damit sind die Ad-Hoc-Änderungen von den Möglichkeiten her sehr beschränkt. Semantische Systeme und Systeme, die mit Abhängigkeiten operieren, haben das Problem, dass der Ablauf mittels einer Erfüllbarkeitsabfrage vorberechnet wird und dann zur Ausführung gebracht wird. Das Ergebnis der Prozesssynthese steht also bereits fest und erlaubt nur noch vordefinierte Eingriffe.

1.3.2 Stabile Systeme

Ein System ist *stabil*, wenn es aus jedem auf ihm definierten Zustand mit relativ wenig Mühe in einen regulären Funkionszustand gebracht werden kann [Dij74]. Was auf jeden Fall nicht passieren sollte ist, dass ein System zu irgend einem Zeitpunkt die reguläre Funkion nicht wieder herstellen kann.

Da ein stabiles System die Fehlerfreiheit dieses Systems impliziert, ist es utopisch zu verlangen, dass ein System in allen Fällen stabil ist.

Was man hier aber verlangen kann ist, dass ein System mit vielen internen Zuständen möglichst leicht für die beteiligten Rollen zu verstehen ist. Auf diese Weise kann man die Fehlerursache gut lokalisieren und, bei ausreichenden Eingriffsmöglichkeiten, auch korrigieren. Diese Korrekturen sollten auch im allgemeinen gut durchzuführen sein, wenn das verteilte System seine Arbeit an anderen Stellen, die gerade nicht von Fehlern betroffen sind, verrichtet. In großen Unternehmen ist es nicht akzeptabel, dass eine ganze Firma steht, nur weil eine Abteilung gerade Probleme zu beheben hat.

Voraussetzung für einen stabilen Betrieb eines kompletten Systems ist also, ausreichend *Entkopplung* zu unterstützen. Die Eigenschaft der Entkopplung erreicht man dadurch, dass ein System keine globalen Zustände zulässt, von denen die *Prozesssteuerung* abhängt. Natürlich ist, aber auf der anderen Seite, möglich, dass eine ganze Firma von einer Reihe an *Daten* abhängt und die fehlenden Daten im ungünstigen Fall den gesamten Betrieb zum Stocken bringen.

Eine weitere Voraussetzung für stabile verteilte Systeme ist, mit Redundanz umgehen zu können. Redundanz ist bei sauberer Trennung von Daten und Laufzeit einfacher zu behandeln. Es ist unproblematisch, Daten redundant zu halten. Dafür reichen einfache und bewährte technische Maßnahmen. Der Umgang mit redundanten Berechnungen ist etwas anspruchsvoller. Diese Arbeit befasst sich im späteren Verlauf mit dem Problem, redundante Berechnungen entsprechend kontrollieren zu können.

1.3.3 Minimale und kolossale Systeme

In der Arbeit werden Systeme betrachtet, die eine bestimmte Funktionalität, die in manchen Fällen einzigartig in der gesamten Architektur ist. Bei diesem Ansatz, ist es unerwünscht die Systeme mit Ballast auszustatten, der wenig oder gar nicht benutzt wird.

Für viele Anwendungen, ist es möglich, die Komplexität herauszunehmen, indem man auf die unbenutzten Teile verzichtet. Dies ist jedoch oft schwierig, weil die Systeme voneinander im statischen Kompilat abhängen. Außerdem gibt es

auch den Fall, in dem Dienste nachgeladen werden, weil sie aus Effizienzgründen bereit gehalten werden, aber während des Betriebs nicht benutzt werden. Dies passiert in sogenannten Web-Containern, wo ein Web-Dienst bereit zum Einsatz ist, aber von Benutzern nicht angesprochen wird.

Systeme, die aus sehr vielen Komponenten bestehen, die strikt von einander abhängen und zudem schwierig nachzuvollziehen sind, weil sie aus tief „vergrabenen“ Schichten bestehen, auf die letztendlich der Entwickler wenig Einfluss hat, werden hier als *kolossale Systeme* bezeichnet.

Hingegen sind *minimale Systeme* aus wenigen Komponenten aufgebaut, die tatsächlich alle benutzt werden oder nicht geladen werden, wenn sie nicht gebraucht werden. Sie bestehen aus einem leichtgewichtigen Skelett, das unbedingt zur initialen Funktionalität gebraucht wird, das im Falle des Gebrauchs mit den Teilen ergänzt wird, welche tatsächlich nötig sind. Außerdem spielt hier die Wiederverwendbarkeit von Funktionalität eine Rolle, sodass der tatsächlich geschriebene Quellcode in mehreren Situationen zum Einsatz kommen kann. Hinzu kommt, dass die Konfiguration der Komponenten im System leicht ist und dem Aufwand der Implementierung angemessen. Am besten fallen hier solche Systeme auf, die in sich geschlossen funktionieren. Das bedeutet, dass die Funktionen im System sich gegenseitig ergänzen und zwar mit den Mitteln, die bereits im System vorhanden sind. Zum Beispiel kann die Konfiguration der Web-Dienste im System selbst ein Web-Dienst sein, der sich wiederum selbst konfigurieren lässt.

1.3.4 APIs: Einfachheit oder Generalität?

Ein anderer Aspekt bei der Entwicklung von unterliegenden Komponenten, wie einer Middleware, ist, dass die Entwickler sehr umfassende und generelle Konzepte zur Unterstützung von möglichst weit gefächerter Funktionalität bevorzugen. Der Vorteil dabei ist, dass sie so ein Framework mit einer umfangreich gestalteten und vollständiger API sehr schnell etabliert und in viele Software-Komponenten gerne miteingebunden wird. Der offensichtliche Nachteil ist, dass ein dermaßen breit verwendbar angelegtes Framework an Komplexität gewinnt. Es werden in vielen Fällen sehr simple Lösungen gebraucht, aber durch die Abstraktionen in den APIs ergeben sich sehr vielfältige Kombinationen von Steuerungsmechanismen, die man wahlweise als Entwickler für seine Zwecke geeignet einstellen muss.

Als eine Lösung ist ein Kompromiss sinnvoll. Im Vordergrund sollte stets die einfache Benutzung eines Frameworks stehen, welches auf eine bestimmte Architektur hin zugeschnitten wurde. Wird dem Entwickler mehr Funktionalität

im Framework durch eine Wahl der Implementierung angeboten, sollten stets in der Architektur am meisten gebrauchten Paradigmen bevorzugt behandelt werden und entsprechend für ihn voreingestellt werden. Die Ausnahmen sollen dann mit etwas mehr Aufwand implementiert werden können. Auf diese Weise kommt man relativ schnell zu einer Implementierung einfacher Systemkomponenten, die innerhalb einer Architektur vordergründig betrachtet werden.

Es gibt in der Software-Entwicklung aktuell, den Drang keine schwergewichtigen Standardkomponenten für Lösungen zu benutzen, sondern Komponenten auf eine bestimmte, zugeschnittene Funktionalität zu spezialisieren. Eine solcher sehr bekannten Bewegungen ereignet sich im Bereich der Datenbanken im Zusammenhang mit dem Web. In dem verteilten und entkoppelten Web konnte sich das Paradigma der eventuellen Konsistenz [Vog09] [Bre00] [GL02] durchsetzen und die herkömmlichen relationalen SQL-Datenbanken [CB74], die auf harte Konsistenzbedingungen optimiert worden sind, haben sich als hinderlich für die aktuellen Systeme im Web erwiesen. Es entstand die *NoSQL!*-Bewegung [Edl12], wo die Entwickler sehr sorgfältig darüber nachzudenken begannen, wie stark konsistent die Daten zur Verfügung gehalten werden müssen, ob Relationen tatsächlich nötig sind und ob SQL wirklich die einzige angemessene Lösung ihrer Probleme ist.

Für ein Problem wird hier ein Werkzeug gesucht und notfalls entwickelt, das zum Problem passt, anstatt ein generelles Werkzeug zu wählen, welches die Funktionalität zwar anbietet, aber aus diversen Gründen nicht optimal arbeiten kann. Die Praxis hat gezeigt, dass diese Herangehensweise zu den besten Anwendungen führen kann. Dazu zählen insbesondere die erfolgreichsten Anwendungen im Web, die ohne genau abgestimmte und spezialisierte Werkzeuge gar nicht existieren könnten.

Ähnlich wie mit *SQL* und *NoSQL!*, verhält es sich mit den Workflows und dem Web als Architektur. Es wurde schon oft von Entwicklern bemängelt, dass Web-Dienste mit SOAP (das sogenannte *WS-**-Prinzip[1]) sich mit dem Web gar nicht verträgt [zMNS05]. Das Web ist eine verteilte, entkoppelt arbeitende Architektur, die von Caching profitiert. Die *WS-**-Realisierungen missachten diese grundlegenden Prinzipien, indem sie Kommunikationszustände, verbindungsorientierte Protokolle und Prozesssteuerung mit zentralisiertem Managementkomponenten einführen. Natürlich können *WS-**-basierte Lösungen funktionieren und es gibt sehr wohl sinnvolle Szenarien, die keinen massiven Parallelismus unterstützen müssen, aber sie vertragen sich nicht mit der darunterliegenden Architektur, die hier offensichtlich zweckentfremdet wurde. Wie Prozesse in der Architektur des Webs eigentlich funktionieren sollten, hat bereits Roy Fielding in seiner Dis-

[1] es hat sich eingebürgert den Web-Services Stack nach WS-BPEL, WSDL und SOAP mit *WS-** zu bezeichnen

seration erklärt [Fie00] und diese Motivation ist stets in den einführenden Worten von vielen Papieren zu finden, die sich mit *REST* beschäftigen.

1.4 Fazit zu Prozessarchitekturen

Aus der Welt der Prozesse kann man in verschiedenen Formen positive Eigenschaften ableiten, die für eine neuartige Prozessarchitektur praktisch nützlich sein könnten und eine Motivation für ein Experimental-Framework sind, das im späteren Kapitel 6 vorgestellt wird. Die folgenden Merkmale stechen hier hervor.

Bei den Workflows 1.2.1 ist die Wiederverwendbarkeit von Diensten nützlich, denn als Entwickler will man nicht, dass gleiche Arbeit für ein gleiches oder ähnliches Szenario wiederholt werden muss. Bei der semantischen Modellierung 1.2.2 sind Schnittstellen interessant, die verständlich und in vielfältiger Form benutzbar sind. Bei den Abhängigkeitsmodellen 1.2.3 ist die einfache Art und Weise wie Parallelisierung und Synchronisierung gestaltet ist von Vorteil. Bei den Multiagentensystemen 1.2.4 ist das Paradigma der Parallelisierung und die kooperative Lösung von Problemen ein sehr interessantes Merkmal. Das CEP-Modell 1.2.5 zeigt Stärken in der Integration und Erweiterung mit weiteren Systemen. Und aus den datenorientierten Modellen in 1.2.6 ist die Trennung zwischen Laufzeit, den leicht zu persistierenden Daten mit fest definierten Standorten und der Fokus auf datenverarbeitenden Prozessen und der Abbildung von Zuständen in der Datenwelt ein hervorragender Anfang, um einen Architekturstil aufzubauen.

Aus den anderen Abschnitten kann man zudem mitnehmen, dass die Unterstützung von Menschen einen erheblichen Mehrwert bringt. Außerdem sind flexible, stabile, möglichst minimale und einfache Architekturen zu bevorzugen, die möglichst viele Freiheiten in der Entwicklung lassen, aber sich auf das wesentliche zu lösende Problem konzentrieren.

Kapitel 2

Ziele dieser Arbeit

In diesem Kapitel wird zusammengefasst, was die Ziele dieser Arbeit sind und welche Resultate hierbei erwartet werden, auf die später im Detail eingegangen wird.

2.1 Kernaufgabe

Die Arbeit beschäftigt sich mit dem Problem, elementare Mechanismen zur Steuerung von datenbasierten Architekturen zu finden, welche mit Hilfe von Informationsübertragung betrieben werden. Als gutes Beispiel und eine Instanz solch einer Architektur wird das allseits bekannte *World Wide Web* (WWW) genommen. Der Interessante hierbei ist, dass das WWW mit darunterliegendem Ressourcenmodell vordergründig eine Architektur nach dem Request-Reply-Paradigma ist und damit keine Begriffe wie „Laufzeit“ kennt, wenn es um Prozessgestaltung geht. Wenn Laufzeit nicht vorhanden ist, kann auch allgemein ein Prozess nicht definiert werden. Das schwierige hierbei ist, dass eine solche verteilte und datenbasierte Architektur ausschließlich lokal beschrieben werden kann und globale Beschreibungen weder interessant noch machbar sind.

Um Prozessunterstützung zu ermöglichen, wird in der Arbeit die Steuerung von Prozessen etwas weiter gefasst und neu gestaltet. Es wird gezeigt, dass elementare Datenstrukturen, die in Ressourcen eingebettet werden, durch ihre Eigenschaften für Prozesssteuerung geeignet sind. Um Ressourcen auf diese Weise nutzen zu können, müssen Verfahren gefunden werden, Daten und Informationen von einem Ort an den anderen so zu befördern, dass die Systeme in konsistenten Zuständen gehalten werden können. Die verteilte Arbeitsweise kombiniert mit der natürlich gegebenen eingeschränkten Sicht auf die Laufzeit in der Architektur ist hierbei eine besondere Herausforderung, die man beachten muss. Eine echte verteilte Architektur sollte stets eine gewisse Unabhängigkeit der, in

der Architektur funktionierenden, Komponenten unterstützen und die Abhängigkeiten von zentralisierten Eigenschaften eliminieren.

Die einzelnen Fragen, die hier auftauchen, sind unter anderem:

- Wie transportiert man zuverlässig Informationen auf verteilten datenbasierten Architekturen?
- Wie synchronisiert man den Informationsfluss und sorgt dafür, dass Eingaben und Ausgaben zur Verarbeitung konsistent zur Verfügung stehen?
- Welche Mittel zur Koordination und Steuerung des Informationstransports gibt es?
- Wie geht man mit Parallelisierung im Sinne von Redundanz und Verteilung um?
- Welche Art von Modell kann man für Prozesse in dieser Architektur benutzen?
- Wie verhält sich dieses Modell zu anderen bekannten Prozessmodellen?

Im Fokus dieser Betrachtung stehen vor allem massiv parallelisierte, menschen- und maschinengetriebene Prozesse und ihre Koordination, wie sie aus dem Bereich *kooperative Problemlösungen* stammen, wie zum Beispiel *Crowdsourcing*. Der Charakter dieser Art von Prozessen basiert auf diesen Einzelschritten:

- Große Aufgabe in kleineren Einzelteilen für Bearbeitung geeignet ausgeben.
- Unterlösungen einsammeln.
- Unterlösungen in Ordnungen bringen.
- Zu größeren Lösungen zusammenfügen.
- Gesamtlösung präsentieren.

2.2 Anmerkung zur Kernaufgabe

Im Zentrum dieser Arbeit geht es um Informationsausbreitung mittels Kommunikation und Protokollen auf verteilten ressourcenbasierten Systemen. Es werden hier einige Hinweise gegeben, wie man mit Systemen umzugehen hat, wenn man sie kompositional zusammenfügen möchte, um eine größere Funktionalität in einer verteilten Architektur zu erreichen. Man möge sich ein Bild vor die Augen führen, in dem die Architektur vorrangig datenbasiert ist, und in der die

Daten zur Verarbeitung zur Verfügung gestellt werden. Diese Daten und Informationen stellen einen passive Systemkomponente dar und liefern einen Entkopplungsmechanismus zwischen Systemen. Diese greifen gemeinsam darauf zu und liefern weitere Daten, die weitere Systeme ansteuern.

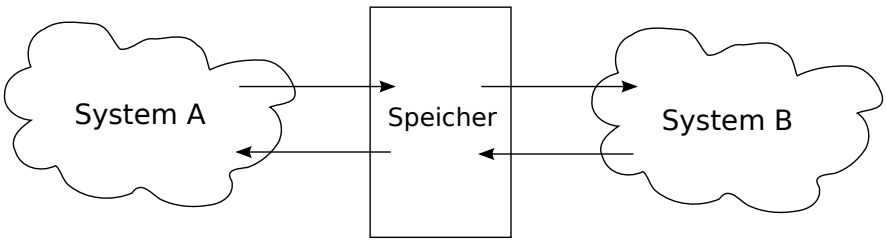

Abbildung 2.1: Datenbasiert entkoppelte Systeme in einer verteilten Architektur

Die Abbildung 2.1 zeigt zwei Systeme, die über eine datenbasierte Komponente kommunizieren. Da die beiden sich nicht gegenseitig beeinflussen, müssen sie auch keine Annahmen über jeweils das andere System machen. Sie sehen nur den Speicher, über den sie kommunizieren. Wichtig dabei ist, zu erkennen, dass das hier eine sehr stark vereinfachte Abbildung ist und sowohl der Speicher als auch die Art, wie der Speicher benutzt wird, bestimmten Prinzipien gehorchen muss. Es ist wichtig anzumerken, dass die hier geschaffene Entkopplung nicht als gemeinsamer Speicher („shared memory“) verstanden wird, welcher es erforderlich macht, dass Systeme Zugriffe exklusiv gestalten müssen. Die weiterführende Terminologie und die Prinzipien des Zugriffs auf die datenbasierten Komponenten werden später im Kapitel 7 eingeführt.

Insbesondere werden hier zwei wichtige Prinzipien eines verteilten datenbasierten Systems vorgestellt und deren interessante Eigenschaften untersucht:

- Idempotenz [Pei81]
- Monotonie

Im Laufe der Arbeit stellt sich heraus, dass diese beiden Eigenschaften wichtig sind, um Systeme zu steuern, die verteilten Informationsfluss unterstützen. Sie werden deswegen formal betrachtet und später anhand von Beispielen illustriert.

Um die Arbeit nicht zu abstrakt zu gestalten, wird stellenweise das *World Wide Web* als eine Instanz dieser Art von Architektur angegeben. Jedoch ist dies alles sicherlich etwas genereller anzusehen und sollte nicht auf diese einzelne Architektur eingeschränkt werden.

2.3 Web-Architektur und Ausführung von Prozessen

Im Kapitel 3 wird auf Eigenschaften und Probleme von herkömmlichen Workflow-basierten Architekturen eingegangen und einige Hinweise gegeben, was problematisch in diesem Zusammenhang ist. Anschließend wird ein Vorschlag für einen Architekturstil gegeben, der die Parallelisierung als Standardmechanismus zur Kontrolle von Prozessen favorisiert. Dieser Architekturstil muss zunächst formal beschrieben werden, damit, darauf aufbauend, der Umgang mit dieser erklärt werden kann.

Während der Arbeit an dieser Idee, sind weitere Gedanken von Experten auf diesem Gebiet miteingeflossen, die als vorteilhaft erkannt worden sind. Ein Architekturstil, welcher hier einen größeren Einfluss auf die spätere Entwicklung und Anpassung der verteilten Architektur hatte, war das Paradigma, nachdem das *WWW* entwickelt worden ist, *Representational State Transfer* (REST), welches im Absatz 5.7.1 weiter motiviert wird.

Als Resultat der Analyse wurde eine Architekturstil entwickelt, der aus zwei Arten von Komponenten besteht, aus *Ressourcen*, wie man sie aus *REST* (siehe auch 5.7.1) kennt und *Agenten*, deren Verhalten im späteren Verlauf der Arbeit weiter untersucht wird (Kapitel 8).

2.4 Datengetriebene Prozesskontrolle auf ressourcenbasierten Architekturen

Der besondere Charakter der behandelten Architektur klärt sich erst, wenn man versucht zu verstehen, wie Prozesse auf einer „inaktiven Datenwelt“ aus Ressourcen unterstützt werden können.

Als naheliegendstes Problem entsteht hier der Aspekt des Informationsflusses zwischen passiv gestalteten Ressourcen. Es ist nicht unbedingt klar, wie man so etwas wie Kontrolle (oder auch Steuerung) in so einer Art von Architektur ausüben kann. Hinzu kommt, dass Architekturen, die hier behandelt werden, im Gegensatz zu herkömmlichen Workflow-Systemen, das Paradigma der parallelen Verarbeitung in den Vordergrund stellen. Diese zusätzliche Bedingung führt dazu, dass die Parallelisierung als Standardmechanismus zur Prozesskontrolle zuallererst einen höheren Aufwand in eine Architektur bringt.

Während herkömmliche Workflow-Architekturen wegen dieser Komplexität die zentralisierte Steuerung (*Orchestrierung*) als Lösung wählen, ist in verteilten ressourcenbasierten Architekturen, die Kontrolle anhand von lokalen und dezentralen Eigenschaften zu gestalten. Diese Steuerungsform kennt man auch unter

dem Namen *Choreographie* [Pel03].

Natürlich ist die Beschreibung der Choreographie in einer verteilten Architektur aufwändig, aber hier ebenfalls notwendig. Die (unmögliche) Alternative wäre an dieser Stelle das Verhalten der kompletten datenbasierten Architektur (wie zum Beispiel des *WWW*) zu orchestrieren, was dem Sinn der Entkopplung in der Architektur widersprechen würde.

Mit dem konkreten Beispiel des *World Wide Webs* sind also Mittel auf den Tisch gelegt worden, mit welchen Problemstellungen mit zentralen Prozessbeschreibungen nicht lösbar sind. Jedoch ist das Web potenziell eine der größten Architekturen, die wir kennen und unterstützt entkoppelte Rechenkraft in vielerlei Formen. Diese Art von Architekturen von der Prozesssteuerungsseite einsatzfähig zu sehen, ist sicherlich erstrebenswert.

2.5 Datenstrukturen als Primitive zur Synchronisierung

Das Paradigma der Parallelisierung als Standardmechanismus innerhalb des Architekturstils bringt neue Eigenschaften mit sich. Früher oder später möchte man Interaktion zwischen Systemen in einer konkreten Architektur modellieren und dazu müssen die einzelnen Systeme zu einem gemeinsamen Ziel gelangen (Konsens). Neben den altbekannten Methoden, durch *Sperrungen* für *Konsistenz* zu sorgen, wird außerdem nach neueren Methoden gesucht, die sich mit der *Entkopplung* der Systeme besser vertragen.

(Abstrakte) Datenstukturen, die in Form von *REST-konformen* Ressourcen angeboten werden, sind eine geeignete Methode, um Informationsfluss im angestrebten Architekturstil zu verwalten. Es zeigt sich, dass solcher Formalismus Ordnungen in eine verteilte datenbasierte Architektur einbringt, der im Zusammenhang der Idempotenz- und Monotonieeigenschaften dazu genutzt werden kann, Verteilung und Redundanz zu steuern und effektiv einzusetzen, um größere verteilte Prozesse durch *Synchronisierung* [AS83] [AHV85] zu unterstützen.

Das Problem der Synchronisierung besteht darin, Daten in geegneter Art und Weise zur Verfügung zu stellen, um diese in konsistenter Form an weitere Bearbeitungsroutinen in anderen Systemen zu überreichen. Dieses Problem besteht insbesondere dann, wenn es um Kommunikation geht, die datenbasiert gestaltet ist. In diesem Fall, ist es nämlich nötig, die Zugriffe auf Daten entsprechend anzuordnen, sodass als Gesamtwirkung in einer verteilten Architektur das beabsichtigte Ziel anstrebt wird.

2.6 Synchronisierung in verteilten Architekturen

In verteilten Architekturen beschränken sich die Mittel zur Steuerung zumeist auf lokale Eigenschaften, damit ein System effizient, ohne teure globale Zustände, arbeiten kann. Manche Architekturen, wie zum Beispiel das WWW, unterstützen eine globale Synchronisierung nicht. Es ist nicht nur die Größe des Systems, sondern auch die entkoppelte Arbeitsweise der Komponenten, die dafür sorgen dass es einen global beschriebenen, konsistenten Laufzeitzustand nicht geben kann.

Mit Hilfe von Ressourcen und den darin gekapselten Datenstrukturen, ist es möglich lokale Konsistenzbedingungen zu schaffen, um ein verteiltes datenbasiertes System zu betreiben. Wie solche Datenstrukturen aussehen und welche Merkmale sie aufweisen müssen, wird in dieser Arbeit ausführlich erörtert. Dies alles gehört zum Aspekt „Kontrolle“[1], die mit Hilfe von bestimmten Arten von Synchronisierung ermöglicht wird.

Wie Synchronisierung im Detail gestaltet werden muss, um den ressourcenbasierten Architekturstil zu unterstützen, wird aus zwei verschiedenen Perspektiven betrachtet. Erstens geht es um die Ressourcen und ihren Aufbau. Als zweites folgen die Agenten und ihr Verhalten. Es zeigt sich, dass wenn die beiden Komponentenarten in der Architektur zusammen gewisse Bedingungen erfüllen, man genügend Mittel in die Hand bekommt, um Prozesssteuerung zu unterstützen. Diese Mittel werden hier sorgfältig ausgearbeitet und auf ihre Wirkungsweise in dem Architekturstil untersucht. Dies findet zuerst in mehreren theoretischen Teilen statt, die ebenfalls empirisch auf ihre Praxistauglichkeit untersucht worden sind.

[1] Der Begriff der Kontrolle wird in dieser Arbeit mit der Steuerung des Kontrollflusses innerhalb von Prozessen gleichgesetzt.

Kapitel 3

Paradigmen der Parallelisierung in Prozessen

Bereits in den 70er Jahren wurde von Richard J. Lipton die bekannte „theory of movers“ vorgestellt [Lip75]. Dieser theoretische Ansatz diente zur Untersuchung von Parallelisierungseigenschaften in parallelen Programmen. Lipton hat versucht, eine Methode zu finden, mittels syntaktischer Untersuchungen des Codes herauszufinden, ob ein sequentiell zusammenhängender Code-Bereich atomar ausführbar ist. Damit gab schon damals einen entscheidenden Hinweis, wie man mit Transaktionen [GR92] [WV01] [Gra81] [LBK01] und Sperrungen [Hoa78] [Dij71] („locking“) umgehen soll und die Untersuchung bezüglich Korrektheit solcher Programme (Verifikation) wesentlich vereinfacht. Ferner hat er Beispiele gegeben, wie Sperren zu Deadlocks führen können.

Die Idee der orchestrierten Geschäftsprozesse folgt im allgemeinen diesem Schema. Transaktionen, Sperrungen und allgemein Kontrolle sind im Zentrum der Aufmerksamkeit, wenn verteilte Prozesse entwickelt werden sollen. Geschäftsprozesse auf Basis von Orchestrierung arbeiten auf gemeinsamen Ressourcen, auf denen die Zugriffe geordnet und in vielen Fällen atomar gestaltet werden müssen. Synchronisierung ist bei Orchestrierung stets im Vordergrund bei fast allen kontrollflusssteuernden Anweisungen.

Dies fällt bereits auf bei dem einfachsten Konstrukt, der Sequenz. Weil die meisten relevanten Geschäftsprozesssprachen zur Ausführung aus dem Workflow-Bereich, zum Beispiel bei jABC [Nag09], WS-BPEL [OAS07] oder XPDL [Wor08], den Sequenzoperator als Standardformalismus zur Modellierung motivieren. Entsprechend findet man in den Modellierungswerkzeugen als Standardverknüpfung für Aktivitäten den Sequenzoperator.

Im folgenden wird das Paradigma der Geschäftsprozessorientierung, wie sie heutzutage üblich ist, aufgegriffen und auf die resultierenden Eigenschaften hin-

gewiesen. In späteren Kapiteln wird ein anderer Architekturstil vorgestellt, der diese Probleme aufgreift.

Bevor die alternative Denkweise in Form eines Paradigmas zur Ausführung von verteilten datenbasierten Prozessen motiviert wird, sollen einige Aspekte vorgestellt werden, die Kernmerkmale von verteilten Systemen darstellen. In diesem Kapitel werden einige grobe Architekturmerkmale aufgegriffen, die an Prozessorientierung interessant sind und wo Defizite herrschen, bei der Modellierung und der Ausführung. Es wird hier auch versucht, die Zusammenhänge zu konstruieren, warum es zu diesen Nachteilen in den bekannten Architekturen kommt.

Dieses Kapitel dient auch als eine Motivation, mehr darüber nachzudenken, wie man auf natürliche Weise mit der Parallelisierung in einem Architekturstil umgehen kann. Sehr viele Überlegungen dazu basieren darauf, die Parallelisierung bei einem moderneren Architekturstil zu favorisieren, indem man es dort zum Standardparadigma erhebt.

3.1 Sichtweisen auf verteilte Systeme

Der Begriff *verteilte Systeme* variiert in der Literatur [Lam78a] [BG81] [LF81] [Sch02], Wissenschaft und Wirtschaft sehr stark. Um zu verstehen, welche Bedeutung dieser Begriff in dieser Arbeit hat, sollte der Sachverhalt hier zuerst geklärt werden.

Es gibt zunächst zwei Aspekte, nach denen man Verteilung gliedern kann:

1. Architektur des Systems
   - „verteilt“ im Sinne von „netzwerkbasiert“
   - „verteilt“ bezogen auf ein Kommunikationsmodell (zum Beispiel: *Client-Server*)
   - „verteilt“ bezogen auf die öffentliche Zurverfügungsstellung eines Dienstes
2. Organisation des Daten (oft auf Datenspeicherung bezogen)
   - „verteilt“ im Sinne von „repliziert“
   - „verteilt“ bezogen auf Datenzugriff und Datenhaltung (zum Beispiel: *Load-Balancing*)

Bei sorgfältiger Entwicklung verteilter Systeme muss man an dieser Stelle jedoch noch stärkere Prinzipien durchsetzen, wenn es um Verteilung geht. Deswegen wird an dieser Stelle noch ein dritter Aspekt hinzugenommen.

3. Ausführungskontrolle (oder bei Prozessen: Management)
    - „verteilt“ im Sinne von „entkoppelt“
    - „verteilt“ im Sinne von „dezentralisiert“

Es ist nicht immer möglich, eine Architektur zentral zu steuern. Das trifft zum Beispiel auf das *World Wide Web* zu. Es gibt hier keine zentrale Ausführungskomponente, die den Systemen im *WWW* Laufzeit zuordnet. Das WWW wird hier deswegen als *dezentral gesteuert* [Dij74] bezeichnet. Die Systeme im *WWW* arbeiten selbständig und haben im Allgemeinen keinen Einfluss aufeinander. Das wird hier als *Entkopplung* bezeichnet. Solange Entkopplung herrscht, muss keine *Synchronisierung* von Datenbeständen durchgeführt werden. Es muss keinerlei *Kontrolle* ausgeübt werden, um kausale Zusammenhänge zu garantieren.

Was dies alles bedeutet wird in den nachfolgenden Teilen hier im Kapitel weiter ausgeführt. Zunächst werden allerdings die Merkmale von Prozessarchitekturen, wie sie heute zum Einsatz kommen besprochen und die Konsequenzen auf die Entwicklung und die resultierenden Implementationen dargelegt.

3.2 Das Standardparadigma Sequenzoperator

Der Sequenzoperator hat den Vorteil, dass der Ablauf von Aktionen in einem Geschäftsprozess sehr einfach zu begreifen ist. Die Aktion B wird nach der Aktion A ausgeführt (siehe Abb. 3.1).

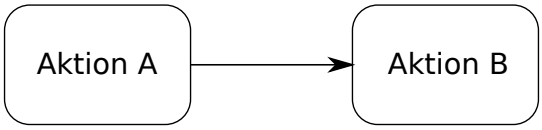

Abbildung 3.1: Sequenz von Aktionen in einem Geschäftsprozess

Dem Entwickler wird hier versteckt viel Arbeit abgenommen, was grundsätzlich gut sein kann, wenn er beabsichtigt, tatsächlich die Sequenz zu benutzen. Die Voraussetzung dafür ist natürlich der Nachweis, dass Aktion B in keinem Fall früher angefangen werden kann, als wenn Aktion A schon beendet worden ist. Dies festzustellen, ist nicht einfach und kostet Zeit bei der Analyse des Prozesses. Der Kompromiss für den Entwickler besteht darin, einfach anzunehmen, dass die Aktionen nacheinander stattfinden sollen.

Bei der Sammlung von Informationen zur Automatisierung eines manuellen Prozesses kommt es vor, dass man eine einzelne Person fragt, wie ein Anwendungsfall aussieht. Und diese erzählt dann chronologisch geordnet was gemacht werden muss, um einen bestimmten Vorgang abzuschließen. Die Erzählform sieht

so aus, dass das Adverb „dann“ gebraucht wird. Dies suggeriert eine feste nicht modifizierbare Abfolge von Aktionen schon in der Prozesserhebungsphase.

Das Entfernen des Sequenzoperators hingegen impliziert in den meisten Fällen eine relaxierte Synchronisierung und wird bei der Prozessentwicklung in der Optimierungsphase angesiedelt [Sch09] (§6.13).

Es sollte klar sein, dass der Sequenzoperator in einem Prozess eine Aktionsabfolge anhand der Laufzeit beschränkt, indem eine zeitliche Abhängigkeit eingeführt wird. Die Formulierung des Prozesses erlaubt nicht, dass man B vor oder während A ausführt. Einen gerechtfertigten Grund gibt es für dieses Verbot nicht. Der Modellierer einer Sequenz sollte sich stets damit beschäftigen, ob das auch wirklich nicht durch äußere Umstände vorkommen kann, dass eine Aktion, wie hier Aktion B, vorgezogen werden kann. Ein Beispiel dafür, dass eine zunächst korrekt aussehende Spezifikation diesbezüglich problematisch ist, wird im Teil 3.9 gegeben.

3.3 Explizit angeforderte Parallelisierung

In Geschäftsprozessen, ausgedrückt in den üblichen Geschäftsprozesssprachen, muss man Parallelisierung anfordern, wenn man sie nachgewiesenermaßen braucht. Dies wird mit einer speziellen kontrollflusssteuernden Anweisung realisiert [vdAtHKB03]. Zusätzlich gibt es aus Einfachheits- und Konsistenzgründen in Geschäftsprozessen Formalismen, die zu den zugehörigen Metamodellen gehören und in denen ein Abschluss der Parallelisierung erforderlich ist. Dies sind entweder diverse *Join*-Operatoren oder es wird implizit angenommen, dass die Parallelisierung am Ende synchronisiert wird, um den Prozessabschluss festzustellen. Der so konstruierte Prozess hat eine Klammerungsstruktur, die den Einsatz von Parallelisierung explizit anfordert und diese auch wieder durch die Synchronisierung der Flusskontrolle in den Ausführungszweigen zusammenführt (siehe Abb. 3.2).

Die Prozessausführungssprache BPEL benutzt für die Parallelisierung das Konstrukt *flow*, welches signalisiert, dass die Ausführung von Aktivitäten ohne Restriktion der Reihenfolge innerhalb des markierten Bereiches abläuft. jABC benutzt in den Modellen eine Semantik, die aus den Prozessen in Unix-Systemen bekannt ist [Ste98] und danach benannt ist, nämlich *fork* und *join*, welche Prozesse abspalten und dann wieder zusammenführen. Da es während dieser Art von ungeordneter Ausführung interessant ist, einige Aktivitäten ausnahmsweise mit einer Ordnung zu versorgen, gibt es bei BPEL das Konstrukt *link*, welches solche Synchronisationsbeziehungen, ebenfalls explizit, ausdrücken kann. Diese

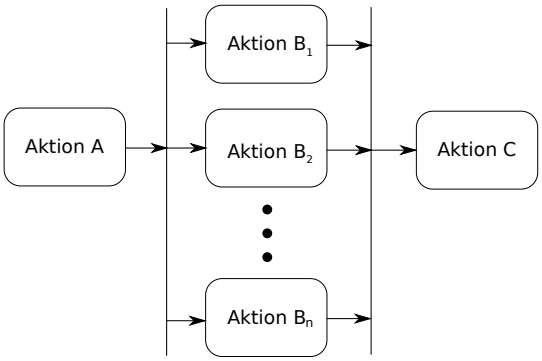

Abbildung 3.2: Parallelisierung von Aktionen in einem Geschäftsprozess

Form der Synchronisation erlaubt es also, Abhängigkeiten zwischen Aktionen festzulegen, wenn sie vorher explizit parallelisiert worden sind.

Bei der Prozesserhebung kommt es zunächst nicht darauf an, ein Unternehmen als eine Sammlung von parallelisierten Abläufen anzusehen. Die Parallelisierung wird viel mehr in der Analysephase als ein Werkzeug zur Optimierung genutzt. Die Tatsache, dass alle Mittel zum Ablaufenlassen einer Aktivität zur Verfügung stehen sind dabei nicht entscheidend, sondern vielmehr der Grund, dass die Parallelisierung vom Geschäftsprozessentwickler dort vorgesehen wurde, also *explizit modelliert* wurde. Die einzelnen parallelen Zweige von Workflows werden von Prozessentwicklern typischerweise als Abfolgen von Aktionen verstanden und verleiten diesen dazu immer wieder implizit die Prozessschritte zu synchronisieren. Auch die grafischen Modellierwerkzeuge, zum Beispiel auf Basis von jABC, BPMN [Obj11] und XPDL, versuchen die Prozessmodelle als verbundene Knotengraphen zu präsentieren, was den Sequenzoperator als Standardwerkzeug sehr hervorhebt und die Parallelisierung schon von der Intuition her als Sonderfall darstellt.

3.4 Ah-Hoc Änderungen der Prozessstruktur

Im allgemeinen besteht die Einfachheit der Geschäftsprozesse (und allgemein auch beliebiger Programme) darin, dass eine allgemeingültige formale Gestalt in Form eines statischen Modells gesucht wird, um ein komplexes Verhalten von Abläufen auszudrücken. Prozesse und Programme mit statischer (Kontroll)Struktur haben den Vorteil, leichter verifiziert werden zu können.

Konzepte für Ad-Hoc-Änderungen [vdABV+99] in Geschäftsprozessen erweitern die statisch formulierten Prozessmodelle um vordefinierte Operationen, die in die Prozessstruktur eingreifen und sie zu einem gewissen grad für einen zu-

künftigen Verlauf abändern können. Dies ist vor allem dann nützlich, wenn ein Geschäftsprozess mit vielen Ausnahmesituationen behaftet ist, oder wenn dieser zum Zeitpunkt der Modellierung nicht vollständig spezifiziert werden kann.

Das Ad-Hoc-Konzept bringt im beschränkten Maße die Programmierung höherer Ordnung auf die Ebene der Geschäftsprozesse. Und es läuft oft darauf hinaus, dass bei Geschäftsprozessen, vielmehr versucht wird parametrisierbare Prozessmuster zu finden, die diese Art von Flexibilisierung erlauben. Die Realisierung von Ad-Hoc-Mechanismen wird oft anhand von Modelltransformationen umgesetzt, die bestimmte temporale Eigenschaften bewahren müssen, die sich wegen der nebeneffektbehafteten Arbeitsweise des Prozessmanagements ergeben.

3.5 Automatische Sequenzialisierung

Definiert man Aktionen als Elemente eines Aktions-Repository, die durch Ein- und Ausgabetypen beschrieben worden sind, ist es möglich diese Typen als Bedingungen für ein logisches System zu formulieren. Für das einfache Konstrukt der Sequenz ist es möglich, eine geordnete Auswahl aus der Menge der zur Verfügung stehenden Aktionen zu treffen, und zwar so, dass wenn ein *Ziel* formuliert wird, dieses Ziel entweder erfüllt wird oder es nicht möglich ist. Dabei darf man die Aktionen in der geordneten Auswahl mehrfach verwenden und die Menge der zu untersuchenden potenziellen Lösungen ist dadurch oft nicht endlich begrenzt, wenn das Ziel unterspezifiziert ist.

Nach diesem Prinzip lässt sich die Bildung von Sequenzen von Aktionen ermitteln. Ob es eine eindeutige Lösung für ein gegebenes Ziel gibt, entscheidet nicht nur die Spezifikationsgenauigkeit des Ziels, sondern auch die Art der Aktionen in der Menge (dem Repository). Das bedeutet, dass der Wiederverwendbarkeitsgrad (wie flexibel die Dienste in verschiedenen Kontexten einzusetzen sind) und die Typenvielfalt (wie viele und welche Typen zur Spezifikation der Schnittstellen benutzt wurden) auf die Zahl der Lösungen einen Einfluss haben. Die Lösung der Frage, ob ein Ziel erreicht werden kann, setzt natürlich dessen Erfüllbarkeit („satisfiability“) mit den gegebenen zur Auswahl stehenden Aktionen voraus.

Bei dieser Art von Sequenzialisierung mittels Inferenz entsteht ein statisches Programm zu einem oder mehrerer möglichen Abläufe. Auf dynamische Aspekte bezüglich der Möglichkeiten zur Ausführung geht diese Form der Inferenz nicht ein.

3.6 Zustandsmanagment und Auswirkungen

Geschäftsprozesse, die aus Aktionen und Diensten bestehen, erfordern nicht, dass diese zustandslos sind. Sogar wenn man Sequenzialisierung gebraucht, ist es möglich, dass interne Zustände gehalten werden, die dann in den entsprechenden Ein- und Ausgabetypen an der Diensteschnittstelle mitberücksichtigt werden müssen. Der Dienst selbst behält aber die Kontrolle darüber, welcher konkrete Typ zur Laufzeit gewählt wird. Die allgemeine Beschreibung der Schnittstellentypen ist allerdings nichtdeterministisch und berücksichtigt alle Möglichkeiten für Typen, die vorkommen können.

Diese Art von Verwaltung erweist sich als höchst komplex, denn ein zustandsbehafteter Dienst kann nicht in einem „Zwischenzustand“ angehalten werden. Dies wiederum wirkt sich aus auf die Ausfallsicherheitseigenschaften des Prozesses und auf die Komplexität des Geschäftsprozessmanagements.

Als Beispiel kann hier ein Dienst genommen werden, der eine Bezahlung mit einer Kreditkarte in einem externen System tätigt („Fire-and-Forget“) und dies lokal im System entsprechend als „bezahlt“ markiert. Um so einen Dienst ausfallsicher zu gestalten, muss es möglich sein, den internen Zustand eines Dienstes zu rekonstruieren. Nach einem Ausfall muss der Dienst feststellen, ob die Bezahlung tatsächlich noch stattfinden konnte, und ob ein konsistenter Zustand („bezahlt“) angenommen werden kann. Man will auf keinen Fall, einen Kunden mit den Problemen rund um eine doppelt gemachte Buchung belästigen. Dies kann durch eine Erweiterung im Dienst des Anbieters der Überweisungsschnittstelle angeboten werden. Dies erzeugt aber im Nutzerdienst weitere Logik mit Entscheidungen und dem Rückgängigmachen von partiellen Aktionen und erhöht seine interne Komplexität.

Das Problem der Zustandsrekonstruktion kann auch auf die Prozessmanagementebene verlagert werden und damit die externe Komplexität erhöhen, durch die Einführung einer Transaktionssemantik. Man führt dazu eine zusätzliche Ebene der Prozesssteuerung ein, die manuell atomare Eigenschaften an bestimmten kritischen Stellen zusichert. Das „Fire-and-Forget“-Beispiel mit der Kreditkarte funktioniert jedoch auch mit Transaktionen nicht, weil das System hier wiederum ermitteln muss, ob im Ausfallszenario die Kredikarte des Kunden belastet worden ist. Hier sind zusätzliche Informationen anhand einer neuen Diensteschnittstelle nötig, um ein System fortgesetzt zuverlässig zu betreiben. Es lässt sich an dieser Stelle nicht mit der synchronen Methode der Laufzeitsynchronisierung arbeiten. Solche Effekte werden einfacher durch Entkopplung mittels datenbasierten Schnittstellen formuliert. In diesem Fall kann ein Prozess zuverlässiger ablaufen, wenn dieser im Status ist, in dem er sich die Bestätigung

mit Hilfe einer Abfrage im Bezahlsystem einholt, bevor die nachfolgenden abhängigen Aktivitäten fortgeführt werden.

Bevor es besprochen wird, wie man ein datenbasiertes verteiltes System durch Steuerung ohne Transaktionen robuster gestalten kann, müssen noch weitere Begriffe eingeführt werden, die System- und Architektureigenschaften beschreiben. Dies folgt später im Teil 5.1.

3.7 Crowdsourcing

Eine ganz ignorierte Eigenschaft im Bereich der Geschäftsprozesse ist Crowdsourcing als Rechenkraft in verteilten Applikationen [Bel09]. Es wird noch einmal im späteren Verlauf der Arbeit auf die Aspekte Redundanz und Lastverteilung eingegangen. Zuvor kann man jedoch durchaus erkennen, dass Crowdsourcing ein Phänomen im Bereich des *WWW* ist, welches, wenn es gezielt gesteuert wird, einen erheblichen Mehrwert Beitragen kann. Das Web ist eine ganz besonders gestaltete verteilte Architektur, die in dieser Arbeit gründlicher untersucht wird. Zusammen mit dem Paradigma der Parallelisierung als Standardmechanismus ergibt sich die Möglichkeit, Berechnungen koordiniert auf Menschen und Maschinen gleichermaßen auszulagern.

Crowdsourcing wird heutzutage dafür eingesetzt, um Arbeitskraft, die in Menschen steckt, günstig zu nutzen. Dazu wird die Masse der Menschen, die am Internet teilnehmen, mit einer größeren Aufgabe in ihren Einzelteilen beschäftigt und dabei eventuell gering entlohnt, wobei das vermittelnde Unternehmen vom Ergebnis der zusammengesetzten Gesamtaufgabe groß profitiert.

Für Menschen, als Arbeitskraft im Bereich Crowdsourcing, ist vor allem das *WWW* als Anwendungsschnittstelle interessant. Dies ist kein Zufall, sondern liegt an dem besonderen Architekturstil, dem das *WWW* vom Aufbau her folgt. Dieser bietet eine einfache Form von entkoppelter Mitarbeit an (Kollaboration an einem gemeinsamen Ziel) und unterstützt mittels logisch und visuell gestalteter Inhalte eine passende Nutzung von Sinneseindrücken. Somit ist es möglich, dass der Charakter der zu lösenden Aufgabe, neben rechentechnisch großen Varianten, auch einfach unlösbar für Maschinen sein kann (Fragen nach Ästhetikmerkmalen und ähnlichem). Hier spielt der Mensch als Teil des Systems eine hervorgehobene Rolle, denn Fragen wie „schön“ oder wie „bedeutend“ ein Sachverhalt ist, sind mit maschinellen Mitteln nur schwierig zu lösen und im Crowdsourcing-Umfeld durchaus üblich.

Seit der Entwicklung des sogenannten „Web 2.0“ [O'R05] stehen auch die benutzergenerierten Inhalte im Vordergrund und fallen unter den Aspekt Crowdsour-

cing [HRW09]. Die massive Parallelisierung von Abläufen, die auf Kommunikation mit Web-Diensten beruhen, erlaubt es dem Betreiber einer solchen Plattform, sehr viele Nutzerdaten zu verarbeiten, die durch Navigation, Erstellung, Verknüpfung und Bewertung anfallen. Da fast der gesamte Benutzerstamm aus Menschen besteht, unterliegt das Verhalten möglichst wenigen Kontrollmechanismen, um die Effizienz des Parallelverhaltens zu fördern.

Crowdsourcing ist lediglich eine spezielle Anwendungsart, die durch eine verteilte Architektur gut unterstützt wird. Diese Anwendungsart wird später im Kapitel 4 verallgemeinert, um Menschen und Maschinen gleichwertig zu behandeln und mit Hilfe des Konzepts von Ressourcen und Agenten zu abstrahieren, worauf die späteren Teile aufbauen.

3.8 Menschen in Prozessen

Die Anbindung von Menschen in Prozesse erfordert eine Möglichkeit zur Entkopplung deren Arbeit und, für die Effektivität des Prozesses ebenfalls nötig, einen Weg parallelisierte Mitarbeit zu integrieren.

Wo Menschen an Prozessen mitarbeiten steigt die Gefahr, den Prozess nicht mehr mit formalen Mitteln beschreiben zu können. Greift der Mensch als Akteur in eine Aktion ein, die innerhalb eines fest definierten Prozesses erwartungsbedingt automatisch abläuft, ergibt sich eventuell ein nicht definierter Zustand. So ein nebenläufig induzierter Zustandsübergang schließt oft externes Wissen ein, das außerhalb des Prozesses von einem Menschen wahrgenommen worden ist. Es kann sogar dazu führen, dass ein zuvor korrekt geplanter Ablauf nicht weiter angewendet werden kann und anschließend eine Inkonsistenz aufweist, weil der Kontrollfluss für so einen Eingriff ins System nicht vorgesehen war.

Dem ist natürlich vorzubeugen. Als Lösung formalisiert WS-BPEL [OAS07] das menschliche Verhalten etwas stärker, indem es den Einfluss auf den Prozess einschränkt. Die Lösung mittels der BPEL4People-Spezifikation [OAS12] erlaubt zwar bedingt entkoppeltes Arbeiten, aber schränkt die Eingriffsmethoden in den Prozess so ein, dass das Ein- und Ausgabeverhalten von Menschen als Teilnehmer am Prozess auf synchrone Art und Weise mit Eingabeschnittstellen zusammenhängt.

An den Schnittstellen des Prozesses für Menschen ergibt sich eine Wandlung vom synchronen zum asynchronen Verhalten, was erforderlich ist, um Menschen zu integrieren. Der rein maschinelle Prozess läuft aus Effizienzgründen synchron ab. Die Aktionen werden in ihrer Kontrolle stark durch das Management gekoppelt. Ist ein Mensch später beteiligt, entkoppelt man die Aktion, indem das

Prozessmanagement in einen Wartezustand geht. Die Aktion eines Menschen induziert wieder Laufzeit in den Prozess, indem dieser eine Eingabe macht, auf die gewartet wird. Der Prozess kann dann wieder synchron ablaufen.

Menschliche Aktivitäten werden deswegen in Workflow-Managementsystemen anders gehandhabt als automatisierte Aktivitäten. Bei Menschen wird asynchron auf ein Ergebnis gewartet und bei automatisierten Aufgaben wird der Ablauf synchron (ohne Wartezeiten) vorangetrieben. Beide Ziele werden jedoch in einem synchronen Konzept der Workflows verwirklicht.

3.9 Laufzeitentkopplung

Zwei Vorgänge in zwei Systemen sind entkoppelt, wenn eine Zustandsänderung während des einen Vorgangs keinen Einfluss auf den anderen Vorgang hat. Zum Beispiel sind zwei Vorgänge entkoppelt, die miteinander nicht kommunizieren und keine gemeinsamen Annahmen über die Ereignisse innerhalb des jeweils anderen Systems haben. Hingegen sind zwei Vorgänge, bei denen sich eine Aktion, in den Vorgängen selbst oder in anderen externen Vorgängen, auf eine Entscheidung des jeweils anderen Prozess auswirkt miteinander gekoppelt oder, bei wenigen solchen Interaktionen, auch *lose gekoppelt* [vdA00].

Enttäuschend dabei ist, dass zwei Systeme in der Praxis oft gekoppelt sind, ohne dass man es beabsichtigt hat. Alleine eine universelle Annahme wie die globale Zeit führt dazu, dass zwei Vorgänge einen Einfluss aufeinander haben. Wenn zwei Vorgänge lesend auf ein gemeinsam genutztes Datum zugreifen, wird das, von der physikalischen Seite her, zu Latenzen führen. Konsequenterweise sind hier die Systeme durch ihre Nebenwirkungen in der Praxis gekoppelt, obwohl sie nur lesend kommunizieren.

Man braucht aber glücklicherweise Entkopplung oft nicht so streng zu definieren. Das Prinzip des World Wide Webs ist, dass zwei Leute unabhängig voneinander ihre Systeme betreiben können, ohne dass der eine auf den anderen angewiesen ist. Man möchte nicht den Fall haben, dass die Probleme und Fehler auf einem System, das andere System stören.

Entkoppelte Systeme findet man meistens dort, wo größere Unternehmen Dienste zur Verfügung stellen, aber diese von den eigenen Betriebsabläufen getrennt funktionieren können. Diese Dienste entkoppeln die unternehmensinternen Abläufe von der restlichen Welt, um reibungslos zu funktionieren. Niemand will ernsthaft sein eigenes System gefähren, indem er unnötige Systeme mit Laufzeitkopplung (also synchron) anbindet. Das bedeutet, zum Beispiel, dass eine Wertpapierbörse die Kurse für Aktien eher nicht für die aktive Verteilung von

aktuellen Börseninformationen zuständig sein will, auch will sie nicht Kunden an interne Systeme koppeln, weil mit steigender Zahl von Systemen die Gefahr von Verklemmungen und allgemeinen Fehlverhaltens steigt. Was sich hier als besser erweist, ist die Informationen in Form von Daten den Kunden zur Verfügung zu stellen, die sich diese bei Bedarf abholen. So können zwei Systeme verbunden werden, ohne sich gegenseitig zu beeinträchtigen. Hier findet eine Kommunikationsform auf Daten statt und sie kann durch einfache Maßnahmen effizienter gestaltet werden, weil datenbasierte Systeme leichter redundant und skalierbar zu gestalten sind als Systeme, die laufzeitgetrieben funktionieren.

Eine wesentliche Problematik stellt die Entkopplung der Laufzeit in Workflow-Systemen. Wie schon erwähnt, ist die synchrone Bearbeitung von Prozessen ein Nachteil in Systemen, sie weitere Faktoren in der Umgebung in Betracht gezogen werden müssen. Dies fällt insbesondere auf, wenn äußere Umstände mit dem Prozess interagieren, die man nicht eingeplant hat oder unzureichend beachtet hat.

Als Beispiel kann hier ein Ausfall einer kompletten Prozessinstanz angegeben werden. Sind für bestimmte Aktionen Fristen festgelegt worden, tritt unter Umständen der Fall ein, dass die Fristen bei der Wiederaufnahme des Betriebs verstrichen sind. Es gibt in diesem Fall keinen Grund mehr den Prozess fortführen zu wollen. Hierbei ist auf eine wichtige Eigenschaft zu achten, dass obwohl der Prozess, aus der Sicht der Laufzeit, steht, trotzdem Änderungen, in den zu berücksichtigten Zuständen, passieren. Diese Zustandsänderungen führen in schlechtestem Fall dazu, dass ein Prozess, der fortgeführt wird, eine falsches Verhalten aufweist und zu einem Fehler führt, der schlecht nachvollziehbar ist, weil er auf impliziten, nicht weiter formulierten Gegebenheiten beruht.

Bei der Entwicklung eines Workflows besteht also die Annahme, dass es keine äußeren Einflüsse gibt, die diesen in einen inkonsistenten Zustand bringen können. Dabei missachtet man, dass all die Zustände, die in der Welt als Einflüsse existieren, gar nicht methodisch erfassbar sind. Im Nachfolgenden wird ein Beispiel gegeben, wie ein kleiner Eingriff in die äußere Welt sich nicht nur auf den Zustand, sondern auf die (eigentlich statisch vorgegebene) Prozessstruktur auswirkt.

In Abbildung 3.3 sieht man einen einfachen Bestellvorgang, den man typischerweise in Geschäftsprozessen sieht. Der Entwickler dieses Prozesses hat offensichtlich eine Entkopplung der Laufzeit festgestellt, die zwischen dem Bezahlvorgang, dem Lernen aus dem Buch und dem Schreiben einer Rezension existiert. Im Allgemeinen wäre das tatsächlich ein ordentlich aussehender Geschäftsprozess, da er Konsequenzen aus vorhergehenden Aktionen betrachtet. Dass der hier angegebene Workflow nicht genügend von möglichen Effekten ent-

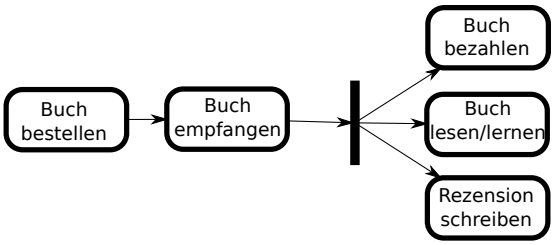

Abbildung 3.3: Ein einfaches Beispiel für einen Bestellvorgang eines Buches.

koppelt ist, sieht man in folgender Abbildung 3.4.

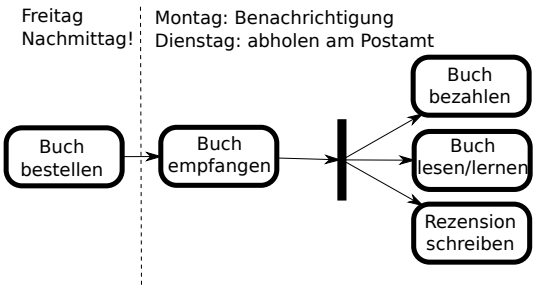

Abbildung 3.4: Äußere Annahmen haben oft keinen direkten Einfluss.

Der bestellende Kunde, als Prozessteilnehmer, stellt in diesem Beispiel fest, dass zur Zeit der Bestellung bereits Freitag Nachmittag ist. Das wiederum bedeutet für ihn, dass er frühestens am Montag eine Benachrichtigung von der Post über eine Lieferung bekommt und am Dienstag erst das Buch empfangen wird. Erst dann kann er das Buch bezahlen, daraus lernen und eine Rezension schreiben. Dies ist natürlich nicht zufriedenstellend, weil die Zeit während des ganzen Wochenendes verloren ist.

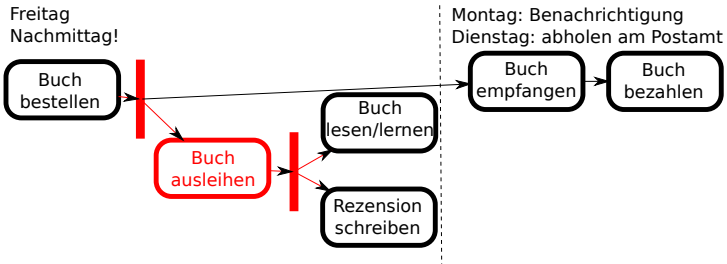

Abbildung 3.5: Strukturänderung des Worklflows bei Einwirkung von äußeren Umständen.

Der Schluss daraus ist ziemlich einfach für den Bestellenden. Er leiht sich das Buch bei der örtlichen Bibliothek aus. Für ihn sollte der Prozess nun ganz anders aussehen (siehe: Abbildung 3.5). Im alten Prozess aus Abbildung 3.3 hat der

Entwickler angenommen, dass eine Abhängigkeit zwischen dem Empfang des Buches und dem Schreiben der Rezension gehört. Er hätte dies vielleicht sogar einplanen können, aber dass man ein Buch schon vorher lesen kann, bevor man die Bestellung erhält, kann man nicht unmittelbar erkennen.

3.10 Synchronisierung der Prozesskontrolle

Fast jede Anweisung in einem Workflowmodell trägt zur Synchronisierung der Kontrolle im Prozess bei. Die Sequenz sagt zum Beispiel aus, dass die Nachfolgeaktion unter keinen Umständen ausgeführt werden kann, bis die Vorgängeraktion ihre Arbeit abgeschlossen hat. So eine Abhängigkeit kann zwar beabsichtigt sein, aber muss nicht unbedingt richtig sein, wie man im letzten Absatz sehen konnte. Sogar die explizite Parallelisierung ist in den meisten Workflow-Modellen gleichzeitig eine Synchronisierung, denn sie wird oft zusammen mit dem Zusammenfluss von Ausführungszweigen modelliert.

In den Persistenzmechanismen eines *WFMS*s werden solche Synchronisationspunkte auf interne Verwaltungsdatenstrukturen abgebildet, die den Kontrollfluss nachvollziehbar machen. Eine typische Sicht auf Workflows ist, dass ein Prozess mittels eines Tokens (einer Markierung) betrieben wird, um den Zustand der Ausführung festzuhalten. Die Tokenposition entscheidet über den Laufzeitzustand des Prozesses und macht (auf rein semantischer Basis) implizit Annahmen über die bestehenden Nebeneffekte im System, an die der Prozessentwickler an dieser Stelle gedacht hat. Die Prozesskontrolle richtet sich also sehr strikt anhand dieser Zustände und resultiert in einem synchronisierten Ablauf. Dieses Verfahren wird *Orchestrierung* genannt und verdeutlicht, dass der Prozess von einem zentralen *WFMS* betrieben wird.

Der aktuelle Prozesszustand funktioniert, ähnlich wie beim deterministischen endlichen Automaten (*DFA*), so, dass zulässige Zustandsübergänge definiert sind. Die Zustandsübergänge sind typischerweise Aktionen oder Kontrollflussanweisungen, die wiederum in einen Nachfolgezustand transferieren. Für diese zustandsbasierte Steuerung muss den Zustand präzise und treffend beschrieben sein und zwar möglichst so, dass weitere Effekte im System die Konsistenz des Zustands einer Prozessinstanz nicht beeinflussen.

Der Zustand eines *WFMS*s ist gekoppelt an bestimmte Annahmen, die der Entwickler berücksichtigt hat. Diese Kopplung alleine verursacht eine starke Abhängigkeit von mindestens zwei unabhängigen Aussagen über das System. Erstens wird der Zustand explizit benannt und zweitens gibt es mindestens eine Annahme, die der Entwickler dem System in diesem Zustand semantisch gegeben hat. Nun kann folgendes passieren:

1. Der Zustand des Systems kann ungültig werden. Das passiert, zum Beispiel, wenn eine Workflow-Engine, einen Fehler aufweist oder eine Workflow-Instanz einfach unvorhergesehen abbricht, zuvor aber den ungültigen Zustand persistiert.

2. Eine Annahme im System ändert sich oder wird hinzugefügt. Dies kann bei unvorhergesehenen Abhängigkeiten passieren oder im Verlauf eines Reengineering des Prozesses.

In diesen Fällen braucht der Prozess eine andere Art von Synchronisierung. Die Konsistenz zwischen der Datenwelt und dem Zustand im Prozess ist im eigentlichen Sinne gar nicht gegeben. Das führt wiederum dazu, dass es ohne manuelle Eingriffe sehr schwierig ist, einen Workflow fortzuführen, wenn solche Abweichungen auftreten.

Das Resultat aus diesen Überlegungen ist, dass man Prozesskontrolle zwar ausüben muss, aber diese muss *sparsam* eingesetzt werden. Allgemein kann man sagen, dass im verteilten System zu viele Zustände (auch ungeplante) existieren, um diese effizient kontrollieren zu können. Man hat hier mit theoretisch unbeschränkter Zahl von parallel arbeitenden Instanzen zu tun und, wie man weiß, steigt die Komplexität (hier: Zahl der Systemzustände) mit der Zahl der Instanzen exponentiell. In diesem Fall wäre das Vorgehen besser, wenn man einen *lokalen Zustand* identifiziert, der eine, von der Struktur her, einfachere (also kleine) Aufgabe durchführt und das Resultat wieder synchronisiert in den Kontrolldatenstrukturen abbildet.

Dass dies zu schön und zu einfach ist, kann man in den folgenden Kapiteln erkennen, wo die Prinzipien der verteilten Kontrolle auf datenbasierten Systemen vorgestellt werden, um die es hier in der Arbeit vorrangig gehen wird.

3.11 Transaktionen

Die stärkste Form der Synchronisierung in einem System, welches keine Fairness garantiert[1], ist die Transaktion [Gra81] [BG81]. Die Transaktion verbindet mehrere verteilte Komponenten des Systems, um einen konsistenten Übergang in einen anderen Zustand zu garantieren. Transaktionen sind als „teuer“ anzusehen. Sie blockieren einen Teil des Systems, was globale Auswirkungen haben kann. Oft erscheint dieses Problem, zum Beispiel, in der Form, in der man mehrere unabhängige Datensätze, die gegenseitige Konsistenzbedingungen aufweisen, atomar abspeichern möchte (zusammenhängend, als eine Einheit).

[1] Fairness ist im Falle von verteilten Systemen ein eigenständiges Problem, welches über der Synchronisierung liegt. In dieser Arbeit wird Fairness nicht weiter betrachtet.

Dass die Notwendigkeit von Transaktionen nicht unbedingt gegeben ist für einige Fälle, und dass sich Transaktionen vermeiden lassen, wird ebenfalls später angesprochen. Die Synchronisierung ohne Transaktionen ist eine interessante Eigenschaft in ressourcenbasierten Architekturen, welche eine Erweiterung von datenbasierten Architekturen sind. Diese Eigenschaft ist auch beliebt in verteilten Architekturen, da sie zur *Sperrungsfreiheit* führt, also kritische Sektionen und exklusive Ausführung vermeidet. [AS83]

3.12 Trennung von Kontrolle und Daten

Im allgemeinen Fall können Daten im verteilten System stets lesend und schreibend zugreifbar gehalten werden. Erst durch Synchronisierungsmaßnahmen über mehrere unabhängige Datensätze hinweg, wird man dazu gezwungen, Kontrolle mit Daten zu vereinen. Insbesondere in einem System, wo Kontrolle in Form von Prozessmanagement völlig zum Stillstand gekommen ist, kann und soll man lesend und, wenn nötig, schreibend zugreifen können. Dies sichert die Möglichkeit zum Fortschritt von Prozessen im System. Die allerletzte Möglichkeit, Daten zu verarbeiten, wenn das Prozessmanagement ausfällt, ist dies manuell zu machen. Wo Menschen sich in eine Architektur gut integrieren lassen, sollte man diese Möglichkeit immer in Betracht ziehen.

Daten und Kontrolle werden bei Workflows oft vermischt. Man entscheidet mit Bedingungen über Teile von Datensätzen in Iterationen, obwohl die naheliegende Idee der Indizierung in einer Ressource über den betreffenden Teil des Datensatzes möglich ist. Beispielweise wird lieber über eine Liste von Kleidern iteriert und abgefragt welches davon die Farbe „blau“ hat, anstatt eine Ressource erstellen, die Referenzen zu allen „blauen Kleidern“ hält. In bestimmten Fällen lässt sich die Kontrolle aus dem Prozess sehr gut extrahieren und die Daten vollständig abstrakt zu behandeln.

Trennung von Kontrolle und Daten verbessert die Effizienz. Einerseits *vergrößert* das Verfahren ein System, macht es aber, bei einem sorgfältigen Entwurf, auch übersichtlicher, indem es relevante Teile voneinander logisch trennt.

Dass Daten und Kontrolle getrennt gehören, wird auch deutlich, wenn man bedenkt wie viel einfacher Redundanz auf persistenten Daten zu realisieren ist. Es gibt sehr viele Methoden verteilte Datenbanken zu implementieren. Es ist möglich diese Datenbanken sehr effizient über einfache Kommunikationsprimitive anzusprechen, weil die elementaren Operationen des Erstellens, Abfragens und Löschen eines Datums sehr einfach gestaltet sind [KJA93]. Hingegen ist die redundante Haltung der Kontrolle in Prozessen stets gut einzuplanen und explizit

zu formulieren [AMG+95]. Erste Probleme ergeben sich, wenn in zwei parallelisierten Prozesszweigen auf gemeinsamen Daten gearbeitet wird. Dies führt dazu, dass man Zugriffe ordnen muss (Sperrungen) [Dij71]. Gibt es zudem bei mehreren unabhängigen Datensätzen gemeinsame Annahmen bezüglich Konsistenz, muss eine ähnliche Ordnung in Form von Linearisierung eingehalten werden, was wiederum die Parallelisierung wiederum stört. Des Weiteren haben Prozesse Nebeneffekte, die verwaltet werden müssen, weil die Trennung von Daten und Kontrolle nicht strikt eingehalten werden muss.

In heutigen Systemen hört man gehäuft den Begriff „relaxierte Konsistenz“ [Vog09] [Bre00] [GL02]. Dieses Verfahren entkoppelt die Laufzeit innerhalb der beteiligten Systeme und sorgt dafür, dass eingehende Daten mit Verzögerung verarbeitet werden können. Es ist oft nicht wichtig, dass ein System eine Synchronisierung durchführt, nach jedem einkommenden Datum. Dafür kann es passieren, dass ein System mit einem älteren Datum rechnet, als es vielleicht zu diesem Zeitpunkt wünschenswert wäre. Dies ist aber nicht essentiell für den Gesamtablauf, weil das neu eingefügte Datum eventuell ein lose angekoppeltes System trotzdem später in den erwünschten Zielzustand bringt. Mechanismen, um die *Aktualität* eines Datums zu behandeln, werden in späteren Kapiteln ausführlicher vorgestellt.

3.13 Zentralisierung

Verteilte Systeme nach Lamport [Lam78a] [Lam78b] versuchen oft die Zentralisierung von Komponenten vollständig zu beseitigen. In diesen Systemen sind verteilte Komponenten, wie Knoten oder Agenten, gleichberechtigt und agieren vollständig lokal, ohne gemeinsame Annahmen über das Gesamtsystem zu machen. Dabei führt das Einschwingverhalten des kompletten Systems trotzdem zu einem globalen, erwünschten Effekt. Hierbei werden *Dezentralisierung*, *Parallelisierung* und *Redundanz* als Werkzeuge benutzt, um keine kritischen Komponenten zu erzeugen. Ausfallsicherheit und paralleles Verhalten der Systeme steht hier im Vordergrund.

Im Gegensatz dazu sind bei Systemen, die Abläufe steuern, die Komponenten in den meisten interessanten Fällen nicht gleichwertig. Eine Komponente für einen Dienst kann im Allgemeinen einen anderen Dienst oder andere Akteure nicht ersetzen.

Die Dezentralisierung wird später noch einmal angesprochen, wenn gezeigt wird, wie man mit der hier vorgestellten Architektur mit gezieltem Einsatz von Kontrolldatenstrukturen verteilte Prozesse steuern kann. Außerdem wird

das Thema aufgegriffen, wenn verteilte Transaktionen in zum Schalten in Petri-Netzen behandelt werden (siehe Teil 9.2). Man sollte sich aber zunächst klar darüber werden, dass in verteilten Systemen, ein solches Maß helfen kann, kritische Teile des Systems zu lokalisieren und diese entsprechend mit Redundanz zu versorgen. Dies kann schon in der Entwurfsphase genutzt werden, um Ausfälle in wichtigen Prozessteilen durch Vorausplanung zu vermeiden. Dezentralisierung ist kein direktes Maß für Ausfallsicherheit, besonders in den Architekturen, die hier angesprochen werden. Sie gibt aber Aussagen über Systemabhängigkeiten und die Konsequenzen von Ausfällen.

Zuerst wird illustriert wie ein Maß für Zentralisierung aussehen kann. Das Maß soll statisch sein und sich an Kommunikation zwischen Komponenten orientieren. Es soll nicht laufzeitabhängig sein oder dynamische Aspekte in die Berechnung der Zentralität einführen.

Die Abbildung 3.6 illustriert das bekannte Beispiel des Client-Server-Modells, welches wahrscheinlich die häufigste Architektur ist, die man im Internet vorfindet.

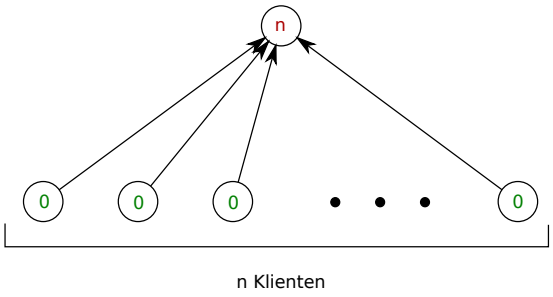

Abbildung 3.6: Zentralisierung beim Client-Server-Modell

In der Abbildung sind Knoten Netzwerk-Software-Komponenten und Pfeile symbolisieren die Abhängigkeit „x *hängt von* y *ab*“ genau dann, wenn $x \rightarrow y$. Die Zahlen in den Kreisen verdeutlichen wie wichtig eine Komponente für die Architekur ist und berechnet sich aus dem Eingangsgrad des jeweiligen Knotens im Grafen.

Man sieht hier also, dass die Klienten einer Client-Server-Architektur nicht wichtig für die Ausfallsicherheit sind, denn keine andere Komponente braucht diese. Ganz im Gegenteil zum Server. Dieser wird von allen Klienten gebraucht und stellt eine kritische Komponente dar.

Etwas verallgemeinert lässt sich die Zentralisierung in einem System abstrakt von den dahinterstehenden Komponenten auf die folgende Art und Weise darstellen (Abbildung 3.7).

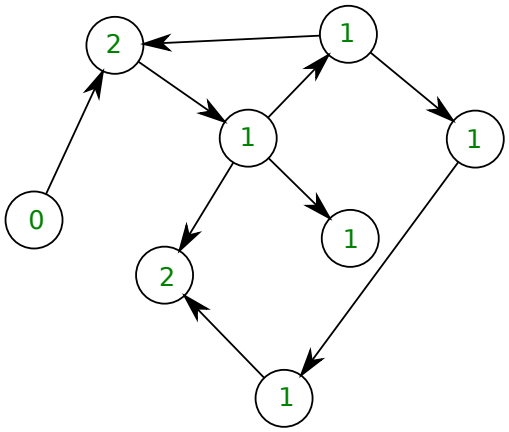

Abbildung 3.7: Knoten eines dezentralisierten Systems

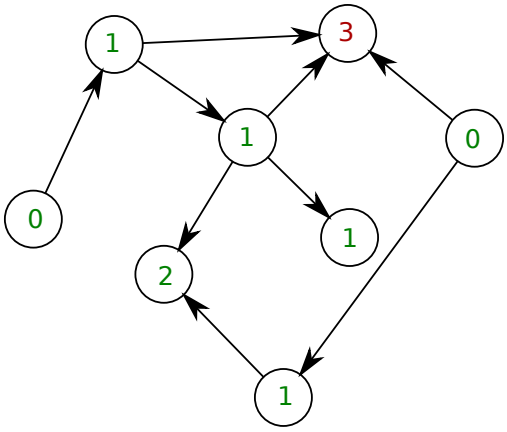

Abbildung 3.8: Knoten eines leicht zentralisierten Systems

Man sieht durch die relativ geringen Schwankungen der Eingangsgrade im Abhängigkeitsgraph, dass die Architektur relativ gut dezentralisiert ist. Modifiziert man einige Abhängigkeiten im gleichen Graphen, dann kann die Zentralisierung etwas anders aussehen. In Abbildung 3.8 sieht man, dass, durch alleinige Umkehr von einigen Abhängigkeiten, die Zentralisierung an einem Knoten etwas deutlicher wurde.

Es wurde schon angesprochen, dass die Dezentralisierung kein Maß für Ausfallsicherheit ist. Denn maßgebend auf die Verzicht auf einen Knoten ist mindestens eine Ersatzkomponente, die die Funktionalität des verlorenen Knotens vollständig übernehmen kann. Außerdem müssen alle eingehenden Pfeile der ausfallenden Komponente auch an die Ersatzkomponente angebunden sein und semantisch die gleiche Art von Abhängigkeiten reflektieren.

Was bringt dann Dezentralisierung, wenn sie in erster Linie nichts mit Redundanz zu tun hat? Dezentralisierung hilft bei Entkopplung von Systemen in heterogenen Architekturen. Wenn der Eingangsgrad eines Knotens hinreichend klein ist, steigt die Wahrscheinlichkeit, dass das gesamte System noch arbeiten kann, da Aktionen im übrigen System weiter entkoppelt durchgeführt werden können.

Man kann hier als Beispiel ein System nehmen (siehe Abbildung 3.9), in dem Nachrichtenschlagzeilen aus verschiedenen Quellen R_Q gesammelt werden und gebündelt in einem Aggregator R_A an Konsumenten R_K gegeben werden. Sollte eine der Quellen R_Q ausfallen, ist der Aggregator zwar betroffen, aber durch die geringe Zentralisierung der Quellen, sind die Konsumenten R_K davon erstmal nicht betroffen. Der Rest des Systems kann also gut unabhängig arbeiten.

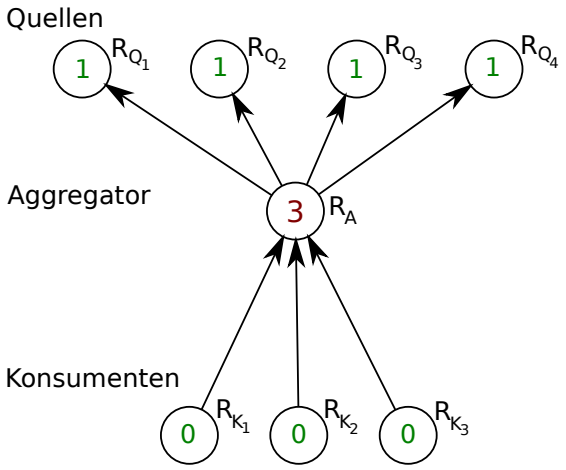

Abbildung 3.9: Beispiel: Newfsfeed-Aggregator

Sollte hingegen eine Komponente mit höherem Grad an Zentralisierung ausfallen, zum Beispiel R_A, sind mehr Systeme betroffen. Es ist zu beachten, dass die Eingangsgrade hier lediglich vereinfachte Aussagen erlauben. Wenn man den Charakter der Information, die an R_{K_j} weiter gereicht wird, mitbeachtet, kann es sein, dass die Konsistenz der zusammengefassten Information aus den R_{Q_i} in R_A wichtig wäre. Ein Beispiel dafür wäre, dass R_A die Aufgabe hat, die die Summe

von gespeicherten Werten in mehreren Ressourcen R_{Q_i}, also $\sum_{i=1,...,4}$ lese(R_{Q_i}) zu berechnen. In diesem Fall würde der Ausfall eines R_{Q_i} die gleiche Auswirkung haben wie der Ausfall von R_A.

Nun hat nicht jede Architektur, die diesem Diagramm entspricht, solche Art von Konsistenzkriterien. Ein Relaxation der Konsistenz in R_A erlaubt hier natürlicherweise höhere Ausfalltoleranz. Entwickelt man solche fehlertoleranten Systeme, kann R_A wiederum den Ausfall einer Abhängigkeit an R_{K_j} weiter geben. Die Entscheidung, ob der Zustand in der Ressource R_A für weitere Berechnungen gültig ist, wird so weiter aufgeschoben. Untersuchungen von Strategien bezüglich Konsistenz werden im Kapitel 8.1 gemacht.

3.14 Kopplung von Systemen

Warum gibt es überhaupt zentralisierte Systeme? Das liegt vor allem daran, dass Architekturen einfach gehalten werden sollen. Man versucht oft Informationen gleicher Art an einem Standort zu halten, um zentrale Systeme zu erhalten, die von allen angekoppelten Systemen abgefragt werden können. Die Wirkung der Zentralisierung ist wünschenswert, denn das gleiche Datum, welches in mehreren angekoppelten Systemen verarbeitet werden soll, an einer zentralen Stelle zur Verfügung gehalten werden kann, anstatt an die einzelnen Systeme einzeln weitergereicht zu werden.

Um jedoch ein zentral bekanntes System zu nutzen, müssen weitere Systeme es kennen und zudem durch Entwickler gekoppelt werden. Diese Reduktion der Komplexität bezahlt wiederum man mit der Erhöhung der Zentralisierung. Um dem Problem eines großen systemübergreifenden Ausfalls entgegen zu wirken, den diese Zentralisierung mit sich bringt, versucht man das zentrale System, das eine Abhängigkeit für eine größere Zahl angekoppelter Systeme darstellt, lokal mit Redundanz zu versorgen.

Also eigentlich dezentralisiert man das System und macht man es wieder komplexer. Allerdings bringt es wiederum den Vorteil, dass Systeme unabhängig von einander arbeiten können, weil sie entkoppelt sind. Hier kommt die Trennung der Daten und der Prozesskontrolle ins Spiel (siehe Abschnitt 3.12). Während Daten möglichst zentral gehalten werden sollten, sollte die Kontrolle möglichst lokal auf das System eingeschränkt werden, innerhalb welchen diese ihre Wirkung hat.

Es ist problematisch, Systeme die mit Daten zu tun haben als reine Datenspeicher anzusehen. Speicher brauchen eine Verwaltung, die wenig Laufzeit in Anspruch nimmt und einfache Formen von Konflikten erkennt. Es wird sich in späteren

Kapiteln herausstellen, dass viele einfache Formen von Konflikten, sich mit Hilfe von *abtrakten Datenstrukturen* lösen lassen, die man bei der Verwaltung benutzen kann. Dies wird dazu führen, dass der Begriff der *datenorientieren Architektur* zu einer *ressourcenorientierten Architektur* erweitert wird.

Wie koppelt man also ein System auf Basis einer bestimmten Architektur, die in der Lage ist Daten und Laufzeit zu trennen und verteilt funktioniert, sodass Störungen eines angekoppelten abhängigen Systems, sich auf andere angekoppelte Systeme möglichst nicht auswirken? Man erlaubt am besten die maximale Parallelisierung und lässt zu, dass Systeme völlig eigenständig entscheiden, zu welchen Zeitpunkten ein zentraler Zugriff passiert.

Eine geeignete Methode dafür ist der *Request-Reply-Mechanismus* [HW03]. Mit diesem Verfahren ist es möglich, die Datenspeicher mit einfacher Logik auszurüsten, um Anfragen durchzuführen und zu beantworten. Die Datenspeicher werden auf diese Weise, logisch gesehen, zu sogenannten *Ressourcen*, die diese Logikschicht beinhalten und einen Teil der weniger aufwändigen Datenverwaltung übernehmen können. Wie die Komponenten genau aussehen, wird im Teil 5.2 behandelt.

3.15 Speicherkapazität der Ressourcen

Die Überlegungen aus Teil 3.14 führten dazu, dass die Laufzeit innerhalb der Ressource, die in einem hochverteilten System zur Verfügung gestellt wird, kostbar ist und sehr eingeschränkt sein sollte. Wie steht es mit der Speicherkapazität?

Der verfügbare Speicher, der Ressourcen zur Verfügung steht, kann immens groß werden, ohne dass die Komplexität nach außen deutlich steigt. Hinter der Ressource, die einen Datenspeicher darstellt mit erweiterter Funktionalität zur Verwaltung, können große Datenbanken oder Speichersysteme stecken, die direkt angebunden sind oder durch weitere Ankopplung intern eine Verarbeitung ansteuern. Des Weiteren können Daten über mehrere Ressourcen, und damit auch über mehrere unterschiedliche Systeme, hinweg verteilt sein. Wie das funktioniert, wird später besprochen.

Einige der später vorgestellten Lösungen für Probleme, die sich in der ressourcenorientierten Architekturen ergeben, sind unter der Annahme gemacht worden, dass der Speicherplatz nicht teuer ist. In der praktisch relevanten Architektur des *WWW* ist dies auch der Fall. Hier gibt es sogar explizit ein Zusammenhang zwischen Qualität eines Systems und der Langlebigkeit von Inhalten hinter festgelegten Referenzen, der in einer Empfehlung namens *cool URI* [W3C98] motiviert worden ist.

3.16 Kurz zusammengefasst

In diesem Kapitel wurden Merkmale von Architekturen analysiert. Einige davon sind wünschenswert und regen zu neuen Überlegungen an (Integration von menschlicher Arbeitskraft 3.8 bis hin zu Unterstützung von Crowdsourcing 3.7, Entkopplung 3.9), andere sollen vermieden werden (übermäßige Synchronisierung 3.2 3.3, Kopplung der Laufzeit unter der Missachtung von unvermeidlichen Nebeneffekten 3.9 3.14). Wiederum andere lassen sich nur durch Kompromisse entsprechend ausbalancieren (Dezentralisierung 3.13, Einfachheit von Redundanzmanagement und Ausfallsicherheit durch Trennung von Kontrolle und Daten 3.12).

Die hier erwähnten Merkmale für Architekturen verfolgen vor allem zwei Ziele:

1. verteilt und entkoppelt (siehe dazu 3.1)
2. ressourcenbasiert nach Request-Reply-Prinzip (letzter Absatz von 3.14)

Beide Ziele führen zu einem Architekturstil, der im Kapitel 5 vorgestellt wird und nachdem sich universelle Architekturen bilden lassen, die für eine bestimmte Art von verteilten Prozessen zu gebrauchen sind. Im nächsten Kapitel werden diesbezügliche Eigenschaften dieser Prozesse untersucht, die sich, durch den natürlich verteilten Charakter, nicht gut als Workflow-Modelle repräsentieren lassen und deren Unterstützung in dieser Arbeit in den Mittelpunkt des Interesses gerückt wird. Es geht hier um sogenannte *kooperative Prozesse*, die sich in *ressourcenbasierte Architekturen* gut einbetten lassen, dadurch dass sie von der sparsam ausgeübten Prozesskontrolle profitieren können. Im späteren Verlauf der Arbeit wird weiter untersucht, inwieweit man auf zentrale Mechanismen zur Prozesssteuerung verzichten kann und welche Konstrukte eine Hilfe bei der früher oder später unverzichtbaren Synchronisierung helfen können, ohne die Prinzipien von verteilten ressourcenbasierten Architekturen zu stören.

Kapitel 4

Kooperative Prozesse auf Daten

Ein *kooperativer Prozess* ist ein verteilter Prozess, an dem verschiedene Akteure kleinere Teile der Arbeit ausführen, um ein Gesamtergebnis zu erhalten [GR04]. Bei dem später vorgestellten Architekturstil wird sich zeigen, dass diese Kooperation nur mit Hilfe von *datenorientierter* Kommunikation möglich und alleine anhand von Daten der Fortschritt feststellbar ist. Ein solcher Prozess ist nicht anhand seiner Laufzeit und deren Steuerung beschrieben und kann dadurch während der Abwesenheit jeglicher Laufzeitkomponente später fortgeführt werden.

Kooperative Prozesse versucht man möglichst synchronisierungsarm zu entwerfen, denn die Synchronisierung stellt eine Restriktion des Parallelverhalten dar. Dabei ist es möglich, dass das Ergebnis insgesamt erst bei einer Synchronisierung des Gesamtsystems in der Architektur erhalten werden kann. Andererseits, kann es sinnvoll sein, das Gesamtsystem weiter arbeiten zu lassen und nur Teile davon betrachtet werden, um eine Teilaussage zu erhalten, die gerade von Bedeutung ist.

Bei solchen datengesteuerten Prozessen steht die Arbeitskraft der einzelnen Akteure nicht konstant zur Verfügung, sondern kann in Form von Aufgabenbeschreibungen ausgedrückt werden, die eine bestimmte Menge von zurverfügungstehenden Akteuren annimmt. Ist kein Akteur bereit, eine Aufgabe zu übernehmen, wird keine Information durch diese (leere) Akteuremenge produziert. Es ist jedoch möglich, dass äquivalente Information aus einer anderen Quelle übertragen wird oder durch andere Akteure an die vorgegebene Stelle hintransportiert werden kann.

Wie man leicht erkennen kann, ist ein solcher kooperativer Prozess, der auf Daten arbeitet, ein sehr flexibler Prozess, aber auch sehr chaotisch, weil recht wenig Steuerung und Synchronisierung als Mittel gebraucht werden. Es ist empfehlenswert, bei manchen Prozessen auf Steuerung zu verzichten, wenn die genaue Beschreibung der Arbeitskraft im Prozess unklar ist. Es ist zum Beispiel nicht

klar, wie Menschen bei einer demokratischen Wahl eine Entscheidung bei einem Wahlvorgang treffen. Es ist vielmehr nicht berechenbar und man möchte davon eher abstrahieren. Ab einem bestimmten Zeitpunkt, müssen jedoch die Einzelergebnisse eines Wahlvorgangs zusammengefasst werden, um das globale Ergebnis zu bekommen. Der Grund, warum man, bei der individuellen Wahl, nur spärlich mit Synchronisierung arbeiten sollte, ist plausibel. Versucht man einen Einfluss darauf auszuüben, ergeben sich Ineffizienzen bei der Parallelisierung. Gibt es zum Beispiel eine Warteschlange bei der Wahl in einem Bezirk, ist diese Art von Synchronisierung plausiblerweise ein Hindernis für die Effizienz des Ablaufs der Wahl. Die Wahl ist ein natürlicher parallelisierter Massenprozess. Sorgt man dafür, dass Synchronisierung spärlich eingesetzt wird, wird der Ablauf insgesamt verkürzt. Theoretisch ist es möglich die Wahl als verteilten Prozess so kurz zu halten, wie ein einzelner für sein Kreuz auf dem Wahlzettel braucht. In der Praxis gibt es jedoch sehr viele, auch unvorhergesehene, Gründe warum es nicht effizient abläuft und warum Synchronisierung nötig ist.

Als Grundlage für solche Prozesse, die diese starre Kopplung aufheben und freie Parallelisierung zulassen, betrachtet man Architekturen, die selbst keine oder wenig Nebeneffekte ausnutzen. Nebeneffekte sind Zustände, die nicht nach außen hin feststellbar sind und trotzdem Aktionen beeinflussen. Möchte man so etwas vermeiden, damit gewisse einfache Garantien gegeben werden bezüglich entkoppelter, verteilter Arbeit, wird man unweigerlich zu Architekturstilen hingelangen, die alles offen zugreifbar speichern und verfügbar halten. Solche Architekturstile und die danach gebauten Architekturen werden fortan *datenbasiert* genannt.

Alle datenbasierten Architekturen, die kooperative Prozesse behandeln, haben eine Gemeinsamkeit. Möchte man eine zuverlässige Aussage über einen verteilten Zustand erhalten, muss die Architektur ganz oder zum Teil synchronisiert werden. Die Verteilung wird damit aufgehoben und zusammengeführt. Mehrere Resultate werden dann nicht zur gleichen Zeit verarbeitet und das entsprechende Ergebnis hängt dann von den Zuständen eines vergangenen Zeitintervalls ab, das wiederum ein inkonsistentes Ergebnis liefern könnte, wenn der Synchronisierungsablauf keinen Beschränkungen unterliegt.

Es ist sicherlich ein Nachteil der datenbasierten Architekturen, dass die Synchronisierung derart schwierig ist. Aus diesem Grund werden im späteren Verlauf der Arbeit *ressourcenbasierte Architekturen* (siehe Teil 5) untersucht, um einen Teil der Synchronisierung zu übernehmen, sodass kooperative Prozesse mit Zuhilfenahme von einigen Regeln bezüglich der Kommunikation schließlich tatsächlich synchronisiert werden können.

4.1 Beispiele für kooperative Prozesse

Die kooperativen Prozesse sind schwierig in eine Form zu bringen, die von Workflow-Engines verstanden werden. Es sind oft Prozesse, die nicht als beendet erkannt werden können, aber das Ergebnis stets zum Greifen bereit liegt. Oft sind die in kooperativen Prozessen behandelten Probleme verwandt mit Optimierungsproblemen. Einige davon haben einen sehr großen Lösungsraum und andere sind einfach nicht berechenbar für Maschinen, weil sie keine formale Wege zur Lösung bieten.

Bei den hier behandelten kooperativen Prozessen handelt es sich um Prozesse, die darauf beruhen, dass vor allem Daten angeboten werden und dann auf Ergebnisse gewartet wird. Die eigentliche Arbeit an den Daten ist meistens entkoppelt. Das bedeutet, man kann nicht viel darüber aussagen ob etwas gerade bearbeitet wird, bis man ein Ergebnis erhält oder wenigstens über eine Aktion informiert wird, die den Fortschritt kennzeichnet.

Die zur Verfügung gestellten Daten können statisch sein und ebenfalls dynamisch, weil sie die Grundlage darstellen für aufbauende Berechnungen. Im Falle von dynamischen Daten, verliert man die Garantie auf Konsistenz. In Teilen 4.1.1 und 4.1.2 sind zum Beispiel vor allem statische Daten zu behandeln. Die ursprüngliche Problemstellung ändert sich nicht und die Daten, auf denen man arbeitet, bleiben konsistent, weil sie nur lesbar sind. Diese Eigenschaft ist auch verwandt mit dem Begriff der *Safety*, welcher im Teil 5.4.4 definiert wird. Bei kooperativen Prozessen mit dynamischen Daten, ändert sich stets die Berechnungsgrundlage. Darunter fällt zum Beispiel ein System wie ein soziales Netz, siehe Teil 4.1.3, in dem Daten ständig modifiziert werden und sich inkonsistente Ansichten für die einzelnen Akteure im System ergeben. Sie arbeiten entkoppelt und können deswegen nicht nachvollziehen, wie aktuell ein abhängiges Datum gerade noch ist, wenn sie ein neues Datum hinzufügen.

4.1.1 Amazons „mechanischer Türke“

Beim Thema Crowdsourcing taucht immer wieder Amazons Crowdsourcing Marktplatz namens *Amazon Mechanical Turk* auf [Mec12]. Die Idee ist simpel und entspricht einem kooperativen Prozess. Eine Firma stellt eine Aufgabe ins Web auf die Mechanical Turk Plattform und überlässt es deren Benutzern (sogenannten HITs; „human intelligent tasks“) diese zu lösen. Amazon behält eine Provision und gibt einen Teil des Geldes welches die Firma bezahlt hat an die HITs. Amazon sichert dabei, dass die Qualität der einzelnen HITs hoch ist, indem die Firma bestimmte HITs mit Fragebögen herausfiltern kann oder ihnen eine Gewichtung zuteilt, die bei der Gesamtlösung berücksichtigt wird.

Hinter HITs verbergen sich natürlich Menschen, daher kommt der Name des historischen mechanischen Türken. Da Menschen abgekoppelt arbeiten und eventuell Lösungen nicht liefern, weil sie unzuverlässig sind, stellen sie eine Komponente im System dar, welche nicht ständig funktionieren muss.

Der Charakter des Prozesses ist der folgende. Eine Untermenge von Aufgaben wird an ein HIT präsentiert, der Benutzer löst diese und schickt die Lösungen an eine zentrale Stelle. Die Firma, die die Aufgabe gestellt hat, kann auf den Bestand der Lösungen stets zugreifen. Einige der Aufgaben werden absichtlich redundant gestellt, damit falsche Ergebnisse durch statistische Methoden von Anfang an ausgeschlossen werden können, um ein präziseres Gesamtergebnis zu bekommen.

4.1.2 Hummeln optimieren Reisedistanzen

Ein interessantes Beispiel für Crowdsourcing, welches Insekten als Akteure verwendet, haben die Forscher an der Queen Mary and Westfield College der Universität London ausprobiert. Hummeln sind in der Lage die Distanzen beim Flug von Blume zu Blume zu berücksichtigen und lösen damit ein schwieriges, mit dem Travelling Salesman Problem (TSP), verwandtes Problem [LCR10].

Die Hummeln sind hier nicht an ein System gekoppelt und liefern stets Teillösungen. Dieses Verhalten lässt sich gut auf Architekturen abbilden, die Rechenkraft als Ressource betrachten, die mal zur Verfügung steht und mal nicht. Die Informationen aus so einem System müssen besonders koordiniert werden und am Ende zusammengetragen werden zu einem Gesamtbild.

Dieses Beispiel zeigt zwei Dinge. Ein verteiltes Problem wird entkoppelt vom eigentlichen System gelöst. Es werden zwar Daten erstellt, aber die Aktionen, um an die Daten zu kommen sind von Komponenten geliefert, die durch Impulse in einer Datenwelt festgehalten werden. Um solche Art von Aktionen formalisieren zu können, braucht man agierende Komponenten, wie Agenten, die in dieser Arbeit später eingeführt werden.

Zum anderen sieht man, dass die Produktion der Information allein nicht direkt zur Lösung führt. Um dieses zu erreichen, bedarf es Konzepte, die verteilt Daten koordiniert abfragen, zusammentragen und weiterführende Ergebnisse liefern. Es ist eigentlich ein vollkommen anderes Problem, die empfangenen Daten auszuwerten, um eine Gesamtlösung zu erhalten. Geht man auch davon aus, dass das System, in dem die Daten eingesammelt werden stets dynamisch ist und global synchronisierbar (zum Beispiel, wenn die Hummeln durchgängig fliegen gelassen werden, um immer bessere Lösungen zu erhalten), dann ergibt sich eine Verwandschaft zu verteilten Systemen, die im World Wide Web arbeiten.

4.1.3 Soziales Netzwerk

Ein soziales Netzwerk ist für sich ein System, in welchem Akteure entkoppelt voneinander arbeiten und Information von anderen Systemen konzentrieren, verarbeiten und in das soziale Netz hinzufügen. Zudem, wenn Information in dieses System eingebracht worden ist, lässt sie sich weiter durch die Akteure verarbeiten, indem diese Verknüpfungen herstellen und diese wiederum ins gleiche System einpflegen.

Ein solches soziales Netzwerk hat eine Eigendynamik, die von den Akteuren angetrieben wird, die immer wieder neue Informationen aus unbekannten Systemen einbringen. Den Akteuren steht nur einen Teil der Architektur zur Verfügung, und das auch nur zu einem bestimmten Momentanzustand. Diese inkonsistente Sicht auf die dort implementierten Systeme ist beabsichtigt und fördert das parallele Verhalten. Viele Architekturen sind in der Lage auf vollständige Synchronisierung zu verzichten und benutzen dazu relaxierte Konsistenzbedingungen.

Daten und Verknüpfung der Daten mittels Referenzen stehen hier im Vordergrund. Was man jedoch auch sieht ist, dass Daten allein nicht ausreichen, um einen solchen Dienst anzubieten, wie ein soziales Netzwerk. Hier sind die Akteure entscheidend und sogar unverzichtbar. Dieses Phänomen der menschlichen Teilnahme an Web-basierten Informationen bezeichnet man auch als „*Web 2.0*“. Denn es sind gerade die menschlichen Interaktionen mit den Daten, die der Web-Applikation (als ein System in der Architektur) erst den Mehrwert geben.

4.2 Merkmale kooperativer Prozesse

Schaut man sich die erwähnten Abläufe an, stellt man fest, dass einige Eigenschaften gemeinsam auftreten. Zunächst, wie schon mehrmals angesprochen, arbeiten hier 2 Arten von disjunkt operierenden Komponenten. Es ist eine datenbasierte „Welt“, die für sich alleine inaktiv ist, dafür aber leicht zu erreichen. Man kann lediglich partielle Ansichten über ihren Zustand abfragen. Im Allgemeinen ist es nicht möglich, die gesamte Architektur konsistent abzufragen. Diese Datenwelt besteht aus mehreren unabhängigen Systemen, die aus Informationsspeichern bestehen, die Daten empfangen und anbieten können und die sich gegenseitig nicht direkt beeinflussen können. In dieser Datenwelt operieren Agenten (in obigen Szenarien als „Akteure“ bezeichnet), die die einzige aktive Komponente sind. Die Agenten brauchen diese Welt, um einerseits miteinander zu kommunizieren, andererseits um Daten zwischen Systemen zu übertragen. Da die Agenten nicht direkt kommunizieren und vom System nicht diskrimi-

niert werden, entsteht resultierend Zustandslosigkeit in der Kommunikation, die solch eine Architektur bezüglich Verteilung von Berechnungen vereinfacht.

Agenten kooperieren also miteinander, ohne direkten Kontakt und ihre Arbeit, obwohl sie für sich allein nur einen kleinen Teil ausmacht, summiert sich zu einem größeren Ergebnis, indem die Systeme, die hierbei relevant sind, die Teile der Arbeit empfangen. Es ist zu bemerken, dass Agenten eine Komponente darstellen, die äquivalent zur Laufzeit in einem Prozessmanagementsystem ist. Sind dort keine Agenten aktiv, bleibt ein System stehen. Eine gute Eigenschaft in dieser Architektur ist, dass ausbleibende Laufzeit die Stabilität der darin funktionierenden Systeme nicht beeinträchtigt. Die Datenwelt ändert sich in diesem Fall einfach nicht.

Von der Verwaltungseite her, gibt es Aufgaben, die mehrfach erledigt werden. Dies kann entweder sinnvoll sein oder es sollte vermieden werden. Dies sollte der Systementwickler entscheiden können. Die Architektur, auf der der Systementwickler arbeitet, muss dafür in der Lage sein, diese Wünsche einfach zu unterstützen, ohne die Verteilung zu stören. Die Prozesskontrolle und das diesbezügliche Zustandsmanagement sollte stets auf der Seite der Ressourcen geführt werden. Die Ressourcen sollten in der Lage sein den Prozessverlauf zu steuern und zwar so, dass die Agenten ermitteln können, was die Ressourcen ihnen für Aufgaben anbieten und wo die Ergebnisse gespeichert werden sollen. Auf diese Weise wird vermieden, dass es *essentielle Agenten* gibt, die unverzichtbar für die Fortführung eines Prozesses sind.

Während die Datenwelt keine Rechenleistung liefert, tun das die Agenten im Überfluss. Dies ist auch der Kerngedanke von Crowdsourcing, wo die Masse die Intelligenz und Rechenleistung darstellt, die man sich nur teuer erkaufen könnte und oft auch nur für einmalige Aufgaben, für die sich eine Anschaffung leistungsfähiger Hardware erst gar nicht lohnt. Es ist auch anzumerken, dass dadurch, dass Menschen (und sogar die Fauna) in eine Architektur integriert werden können, man in der Lage ist, Aufgaben zu lösen, für die keine (guten) Maschinen existieren. Dies zeichnet alle Beispielszenarien in diesem Kapitel als Gemeinsamkeit aus.

Wie herkömmliche Prozesse, können kooperative Prozesse ein Ziel haben oder fortlaufende Arbeiten verrichten. Im Falle von Prozessen, lässt sich explizit für eine globale Steuerungskomponente sagen, welche Arbeiten zu tun sind und wie die Steuerung weiter verläuft. Bei kooperativen Prozessen hingegen, ist diese Formulierung implizit anhand der Systemkonfiguration gegeben. Es ist eine Kombination aus der Restriktion der Möglichkeiten zur Datenorganisation und den zur Zeit verstandenen und vorhandenen Aktionen, die ein Fortlaufen des Gesamtprozesses in eine Richtung lenkt, die der implizit formulierten Aufgabe

entspricht. Dies ist auch im Sinne eines verteilten Systems. Denn eine zentrale Steuerungskomponente, die die Prozesskontrolle verwaltet, wäre hier sicherlich nachteilig für das parallele Gesamtverhalten.

Ein weiterer Unterschied ist, die synchrone Arbeitsweise des Prozessmanagements. Dadurch, dass Prozesssysteme synchron arbeiten und hart verdrahtet sind, reagieren sie auf sich ändernde äußere Umstände nicht. Sie können nicht beurteilen, ob potenziell mehr Arbeitsleistung zur Verfügung steht, sondern planen die Verteilung ein, indem sie im Prozessmodell explizit ausformuliert wird. Dies ist in einer Architektur, wo die Verfügbarkeit von Rechenkraft stets varriert und man davon gerne abstrahieren würde, eher störend.

4.3 Allgemeine Sicht auf das World Wide Web

Die Vorarbeit in den Überlegungen in dieser Arbeit bietet das World Wide Web als Architekturgrundlage. Es ist eine der größten verteilten Systeme. Sie funktioniert basierend auf dem Internet und der Protokollfamilie TCP/IP. Das „*WWW*“ (oder kurz „Web“, wie es sich heute eingebürgert hat) macht Informationen zugänglich in Form von Hypertext, welches in der Lage ist, Informationen miteinander zu verbinden. Die bekannteste Form von Hypertext ist sicherlich die *HTML* („hypertext markup language“) [BL90] [W3C12a], welches zunehmend maschinenlesbarer gestaltet (siehe *XML* [W3C12b] oder *JSON* [JSO12]) wird indem Inhalte von Maschinen gelesen werden können (Syntaxverständlichkeit), aber auch die Bedeutung schon zum Teil erkannt werden kann (Semantikverständlichkeit). Menschen brauchen sicherlich Informationen in anderer Form präsentiert als Maschinen und es hat sich gezeigt, dass das *WWW* Menschen sehr gut integrieren kann. Von einfachen statischen Texten bis zu komplexen Webbasierten Anwendungen, es wird alles für Menschen verständlich formuliert. Vieles ist hier sicherlich eine Sache der Gewöhnung, aber die uneingeschränkte Erreichbarkeit von Web-Anwendungen in der ganzen Welt, macht Dienste, die im Web angeboten werden, ungemein attraktiv.

Den Kern der WWW-Technologie stellen also Daten und Inhalte dar, die in passender Form für den jeweiligen Nutzer in einem bestimmten Kontext dargestellt werden. In späteren Kapiteln werden die Daten zu Ressourcen angereichert, und zwar mit einer zusätzlichen Schicht, die Daten für die Benutzung kapselt und verwaltet. Damit werden Ressourcen im Gegensatz zu der *generischen datenbasierten Ressource* ohne weitere Logik etwas flexibler handzuhaben. Es ist darauf zu achten, dass sowohl solche reinen Datenspeicher als auch Ressourcen Request-Reply-basierte Kommunikation unterstützen müssen. Das impliziert wiederum, dass die Schicht für die Ressourcenverwaltung nicht allzu komplex werden darf.

Die geringe Auslastung von Server-Anwendungen ist an dieser Stelle ein wichtiges Kriterium, um reibungslose Operation zu ermöglichen.

Man kann also feststellen, dass schwierige Berechnungen bei Ressourcen fehl am Platz sind. Schaut man die Architektur des WWW weiter an, gibt es Komponenten, die durchaus mit Rechenkraft und Auslastung zurecht kommen. Es sind die Agenten, die von Nutzern gebraucht werden, um die für sie interessanten Ressourcen zu nutzen. Diese interessante Feststellung führt zu neuen Überlegungen, wie man die Rechenkraft der Agenten in ressourcenbasierten Architekturen für Prozesse nutzen kann. Das wichtigste Mittel, um Agenten zur Kooperation zu bringen, ist elementare Kommunikationsmechanismen zu schaffen. Und da Agenten keine direkte Kommunikation untereinander unterstützen, müssen sie den „Umweg“ über die Ressourcen nehmen. Mit der beliebigen Austauschbarkeit von Agenten und ihrer zustandslosen Arbeitsweise, ergeben sich viele positive Eigenschaften, die funktionalen Charakter haben. Auf der anderen Seite ist das Potenzial durchaus noch größer und bedarf einer sorgfältigen Untersuchung. Im WWW operieren namenslose Agenten, die Aufgaben zwar erfüllen, aber keinem Zwang unterstehen, irgendeine Leistung zu erbingen. Es kann passieren, dass ein System komplett still steht oder das viele Agenten auf einmal das gleiche Ziel haben. Wo in einem parallelen System ähnliche oder gleiche Aufgaben gelöst werden, gibt es die Gefahr von Race Conditions [Cas81] und in den meisten Fällen läuft es darauf hinaus, dass *Sperrungen* eingesetzt werden, um Kontrolle über Zugriffsreihenfolgen zu erlangen. Die bekannten Mittel dazu sind *Semaphoren* [Dij71] und *Transaktionen* [Gra81].

Im Falle des WWW sind in vielen Fällen Sperrungen kontraproduktiv bis teilweise unmöglich. In dieser Arbeit wird die Architektur des *WWW*, mit Hilfe von Ressourcen, einfache Methoden unterstützen, die Sperrungen zu beseitigen. Es ist zwar nicht in allen Fällen möglich, dass Sperrungen vermieden werden können, aber es werden Fälle gezeigt, wo dies funktioniert. Des Weiteren kann mit einigen Werkzeugen in Form der Kombination von Ressourcenmethoden und zugehörigen Protokollen gezeigt werden, dass man in der Lage ist, einige Fälle von Sperrungen etwas passender für ressourcenbasierte Architekturen gestalten zu können.

Kapitel 5

Agentenbasierte parallele Prozessarchitektur

Die agentenbasierte Sichtweise auf das World Wide Web ist nicht neu. Schließlich wird schon in der Terminologie des Webs das Client-Programm als *Browser Agent* bezeichnet. Oft wird in diesem Kontext ein Client-Server-Modell bezogen auf die Kommunikation erwähnt oder das Web als eine Wolke dargestellt, um sich nicht näher mit den innewohnenden Eigenschaften zu befassen.

Unter anderem ist der Zweck dieses Kapitels, das WWW und ähnliche Architekturen anders strukturiert darzustellen, damit das Potenzial zur Prozessausführung auf dieser Architektur deutlich wird. Das Kapitel darf aber durchaus abstrakter zu sehen sein, denn die hier erwähnten Begriffe und Prinzipien gelten gleich wohl für datenbasierte Systeme. Das World Wide Web ist lediglich eine solche interessante Architektur und da diese sehr erfolgreich geworden ist, wird sie in dieser Arbeit zur einfacheren Veranschaulichung benutzt. Genau wie „Representational State Transfer“ ein Architekturstil ist und keine Architektur wie das Web, darf man die Komponenten der Architektur in diesem Kapitel ebenfalls etwas abstrakter sehen.

Systeme, die vorrangig datenbasiert arbeiten und keine direkte Unterstützung für Prozessabläufe anbieten, findet man überall in der Welt. Das bekannteste Beispiel sind Datenbanken und die darauf entwickelnden Anwendungen, wie zum Beispiel ein Repository. Solche Systeme sind in der Lage Daten zu speichern, aber führen selbst kein Management zur Steuerung von Prozessen. Um Prozesse zu unterstützen, ist es wichtig einen Prozesszustand mitzuführen. Natürlich kann und muss man den wiederum in Datenform in einem datenbasierten System abbilden. Die Frage ist aber, ob man das immer zentralisiert machen darf. Öffentliche Architekturen, wie das Web, bestehen aus vielen kleinen Einzelsystemen, die gegenseitig Daten speichern. Steigt die Wichtigkeit eines der Systeme, geht man dazu über, dieses mit möglichst wenigen externen Abhängigkeiten auszu-

statten. Dies gestaltet man am besten, indem man das Pull-Prinzip benutzt und die Kontrolle für Konsumenten der Information vollständig frei gestaltet. Die Information wird nicht mehr verteilt, sondern zur Vefügung gestellt. Daten und Kontrolle werden getrennt in den verschiedenen Systemen gehalten. Das Management wird damit deutlich aufwändiger.

Jemand, der einen datenbasierten Dienst betreibt, hat jede Menge Vorteile gegenüber jemandem der hinzu Prozessmanagement und Kontrolle betreiben muss. Systeme, die Daten ausliefern, sind leichter skalierbar. Sie haben viele einfache Mittel um Last und Speicherung zu regeln. Als Beispiele sind hier Caches [F+99] [Fie00], Redundanzen/Lastverteilung [Rot08] und verteilte Datenbank-Cluster [BG81] bekannt.

Um die Zusammenhänge zwischen Prozessen und einem vorwiegend datenbasierten System besser verstehen zu können, muss man die Sichtweise auf das Web auch etwas präziser fassen. Die hier dazu vorgestellten Begriffe und Mittel dienen als eine Basis für die Terminologie, die in dieser Arbeit gebraucht wird.

Sind die Eigenschaften des datenbasierten Systems und seine Möglichkeiten und Einschränkungen erklärt, ist man erst in der Lage, weitergehende Überlegungen anzustellen zum Thema Prozesssteuerung.

5.1 Ressourcenbasierte Architekturen

Die nachfolgenden Architektureigenschaften behandeln die Prinzipien nach denen ein verteiltes System gebaut werden kann, welches Eigenschaften liefert, die im Kapitel 3 gesprochen worden sind.

Sie entstammen aus den Ideen des Architekturstils *REST* und werden hier weiterentwickelt. Während *REST* über die Ressourcen referiert und ein Paradigma aufstellt, wie mit diesen umgegangen werden muss, damit ein Dienstesystem entsteht, das die Effizienz und Vorteile von mehreren Architekturstilen vereint, wird an dieser Stelle der Fokus auf einen Architekturstil mit Agenten und Ressourcen gerückt. Das Ziel ist, die Begriffe des ressourcenbasierten Architekturstils mit Hilfe von einigen Definitionen einzuführen, um später die Möglichkeiten zur Kommunikation illustrieren zu können und darauf aufbauend das Problem der Prozesssteuerung durch Synchronisierung zu lösen.

Bevor das Thema Synchronisierung angesprochen wird, müssen grundsätzliche Eigenschaften über den hier definierten Architekturstil vorgestellt werden. Später wird mit Hilfe dieser Eigenschaften der Bezug zwischen den Komponenten hergestellt, um einen Überblick über die Funktionsweise einer kompletten darauf basierenden Architektur zu bekommen.

Die hier angesprochenen Architekturen organisieren vor allem Daten in Form von Ressourcen. Ressourcen unterscheiden sich von allgemeinen Datenspeichern dadurch, dass sie zum Teil in der Lage sind, Datenorganisation zu übernehmen. Dabei handelt es sich um relativ einfache und nicht rechenintensive Operationen, die das Request-Reply-Prinzip unterstützen können. Die Menge der Ressourcen in einer Architektur ist eine adressierbare Schicht, mit welcher Agenten kommunizieren.

Die Funktion der Ressource ist also Operationen auf Daten durchzuführen. Die Systeme, die diese Daten bearbeiten von der Laufzeit der anderen Systeme innerhalb der Architektur zu entkoppeln. Auf diese Weise ist es möglich, Datenaustausch zwischen verschiedenen Systemen durchzuführen, ohne dass die Systeme selbst durch Sperrungen und natürlich ohne Dead-Locks gefährdet werden. Ressourcen sind der einzige Weg zur Kommunikation zwischen Agenten. Möchte man eine Architektur als Modell auffassen, ergibt sich ein Bipartiter Graph, der Ressourcen mit Agenten und Agenten mit Ressourcen verbindet. In späteren Beispielen wird, im Falle der Trivialität, ein Agent oft in den Modellen weggelassen und vielmehr der Informationsfluss mit einem Pfeil markiert, um anzudeuten in welche Richtung sich Informationen ausbreiten.

5.2 Grundkomponenten

Im folgenden werden zu allererst die Grundkomponenten des Architekturstils definiert und eingeordnet. Diese Definitionen und Begriffe sind für das Verständnis der späteren Kapitel wesentlich.

Um den Bezug zur Praxis herzustellen, wird am Ende dieses Kapitels die Beziehung zur Architektur des *WWW* erklärt, welches wiederum den Architekturstil namens *REST* miteinfließen lässt. Die Agenten und ihr Verhalten in Systemen sind in *REST* nur sehr oberflächlich aus der Perspektive der Ressourcen beschrieben, welche zum Teil als Dienste verstanden werden. Ressourcenbasierte Architekturen sind in dieser Arbeit abstrakt und kapseln lediglich Daten mit Hilfe von Methoden, die von Agenten aufgerufen und verarbeitet werden können.

5.2.1 Agenten und Agentengruppen

Ein *Agent* A stellt ein abstraktes Konzept dar zur Darstellung einer Komponente in den besprochenen Architekturen, die als einzige *selbständig aktiv* ist. Die Menge der in einer Architektur eingesetzten Agenten wird mit $\mathfrak{A}$ bezeichnet. Sind $A_1, A_2 \in \mathfrak{A}$ zwei Agenten, die das gleiche Verhalten aufweisen, fasst man sie zusammen zu einer *Agentengruppe* $G = \{A_1, A_2\}$. Dieses Verhalten ist entweder for-

malisiert als ein *Agentenprogramm* vorhanden, was einen automatisierten (oder maschinengesteuerten) Agenten auszeichnet, oder spiegelt ein nicht formalisierbare Verhalten wider, zum Beispiel im Falle eines Menschens, der als Agent aufgefasst wird.

Agenten können untereinander nicht kommunizieren. In der besprochenen Architektur wird jeder Zustand, der *essentiell* für die Abläufe im Gesamtsystem ist, in *Ressourcen* gehalten, indem die verfügbaren *Methoden* aufgerufen werden, was im folgenden Teil erklärt wird.

Im allgemeinen sind Agenten in den besprochenen Architekturen nicht anforderbar, um zu einem Zeitpunkt eine bestimmte Aufgabe zu machen. Es ist durchaus denkbar, dass in einem System keine Agenten aktiv sind und kein Fortschritt erreicht wird. In diesem Fall werden in diesem System keine Änderungen auf den zugehörigen Daten durchgeführt. Wie im Teil 3.9 angesprochen, ist die Annahme, dass in der gesamten Architektur keine Änderungen durchgeführt werden eher illusorisch. Schon allein die laufende Zeit fließt unter Umständen in einen Prozess ein und damit kann auf die Entscheidungen eines später erscheinenden Agenten Auswirkungen haben. Außerdem sollte man stets mit Agenten die Daten im System bearbeiten rechnen. Diese können nämlich mehrfach und ohne zeitlichen Zusammenhang auftauchen.

Die aktive Phase eines Agenten lässt sich folgendermaßen beschreiben:

- den Bedarf an Arbeit in einem System *identifizieren*
- die Arbeit *durchführen*
- und das Ergebnis der Arbeit *sichern*

Dazu später mehr im Kapitel 8 über das Agentenverhalten.

Unter der Voraussetzung, dass Gruppen G von Agenten gleichen Verhaltens beliebig groß werden können, ergeben sich Konfliktsituationen, in denen das Gesamtverhalten der Agenten ohne weitere Organisation nicht kalkulierbar wird. Außerdem ergeben sich solche Organisationsprobleme ebenfalls, wenn Agenten ihre Arbeit auf eine konsistente Art und Weise sichern wollen. Dies sind Probleme, mit denen sich die Arbeit in den nächsten Kapiteln beschäftigen wird, nachdem weitere Grundbegriffe eingeführt worden sind.

5.2.2 Ressourcen

Die andere Komponente des Architekturstils sind *Ressourcen*, deren Menge in der Architektur mit $\mathfrak{R}$ bezeichnet wird, und die durch eindeutige universelle Bezeichner adressierbar sind, so genannte *URI*s („universal resource identificator“;

auch im Folgenden Referenzen genannt) [BLFIM09]. Jede Ressource kann mehrere Bezeichner zur Adressierung haben. Die Menge dieser Bezeichner wird hier als $U(R)$ beschrieben.

Jede Ressource $R \in \mathfrak{R}$ hat einen internen Zustand Z. Abhängig vom Zustand Z und von der Nutzung einer bestimmten $u \in U(R)$ bietet eine Ressource R eine Menge von *Methoden* an, $M_{u,Z}$. Jede dieser Methoden $m \in M_{u,Z}$ ist in der Lage auf Z einzuwirken und das Z in den Nachfolgezustand Z' zu transferieren, in dem wiederum eine Menge anderer Methoden, $M_{u,Z'}$ angeboten wird. Ressourcen können untereinander nicht kommunizieren und teilen auch keine Zustände. Die Methoden $M_{u,Z}$, die eine Ressource anbietet, werden ausschließlich von Agenten aus $\mathfrak{A}$ aufgerufen, indem ein Agent die Ressource R mit einer *URI* $u \in U(R)$ anspricht.

Die Referenzen in $U(R)$ sind grundsätzlich *bedeutungsneutral*, jedoch können und müssen einige Ressourcenreferenzen auf $\mathfrak{R}$ global bekannt sein, damit Agenten zu Anfang ihrer Arbeit einen „Einsprungspunkt“ ins System haben, auf welchem sie operieren sollen. Die Neutralität der Bedeutung einer Referenz (URI) stellt sicher, dass man eine Referenz u nicht aus irgendwelchen Daten konstruieren kann, weil sie keinen logischen oder strukturellen Aufbau hat. Weiteres zu diesem Thema folgt im Teil 5.3.2. Diese hier eingeführten Bedingungen basieren auf den allgemein bekannten Überlegungen des „Cool URI“-Prinzips in [W3C98].

Methoden und Methodenaufrufe

Jeder Methode $m \in M_{u,Z}$ können beim Aufruf Daten übergeben werden, die sich auf den Zustand Z der Ressource R auswirken. Im folgenden werden diese Daten beim Aufruf auch *Parameter* genannt. Ein Aufruf einer Methode $m \in M_{u,Z}$ mit dem Parameter p wird nach dem Methodenbezeichner m mit einer runden Klammer angegeben: $m(p)$. Diese Notation des Aufrufs vereinfacht die Erklärungen. Es ist jedoch wichtig, die *URI* als eine Art Namensraum im Kopf zu behalten.

Ist eine Ressource R_1 im Zustand Z_1 und wird eine andere Ressource mittels $u \in U(R_2)$ adressiert, wobei R_2 im Zustand Z_2 ist und $u \notin U(R_1)$ gilt, ändert ein Aufruf $m(p)$ mit $m \in M_{u,Z_2}$ den Zustand Z_1 nicht. Die Gegenseitige Beeinflussung von Ressourcen wird hiermit ausgeschlossen. Allerdings kann ein solcher Aufruf auf einer Ressource eine andere Ressource R_{neu} mit einem Initialzustand Z_{neu} erzeugen, der sowohl abhängig von u, Z_1 als auch p ist. Dieser Vorgang wird Instanziierung genannt und die entsprechenden Konsequenzen werden später im Teil 5.4.2 behandelt.

Eine Ressource R mit $U(R) = \{u, v\}$ kann von verschiedenen Systemen aus mit unterschiedlichen Methodenmengen $M_{u,Z}$ und $M_{v,Z}$ angesprochen werden. Das

Ziel hierbei ist, Berechtigungen zur Nutzung von Methoden durchzusetzen. Es ist jedoch einfacher die Methode m, die ausschließlich von einem System aufrufbar ist, mit einem eindeutigen Bezeichner auszustatten, damit Missverständnisse nicht aufkommen.

Wird eine Methode $m \notin M_{u,Z}$ aufgerufen über die *URI* u und im Zustand Z, wird dem Aufrufenden Agenten ein Fehler signalisiert und der Zustand Z bleibt erhalten. Wird ein Parameter nicht verstanden, wird der Zustand Z ebenfalls erhalten.

Im Falle eines erfolgreichen Aufrufs $m(p)$, wobei $m \in M_{u,Z}$ ist, kann der Zustand der Ressource R von dem aktuellen Zustand Z in den neuen Zustand Z' übergehen. Er kann aber ebenfalls je nach Funktion der Methode m als Z erhalten bleiben. Diese Änderung wird, falls sie interessant für Erklärungen ist, in der Notation $m(p)/[Z \rightarrow Z']$ geschrieben.

Da man nicht verhindern kann dass ein Agent, die Methode $m \in M_{u,Z_1}$ aufzurufen versucht, auch wenn sie im Zustand Z_2 nicht definiert ist, kann die oben angesprochene vereinfachte Form $m(p)$ für den Aufruf benutzt werden. Der Agent hat, aus seiner Sicht, nur beschränktes Wissen über den Zustand und kann diesen im Allgemeinen nicht erfragen. Aus der Sicht der Ressource sind die Zustände jedoch klar und wesentlich für die Steuerung der angebotenen Methodenmengen.

Dem Leser mag aufgefallen sein, dass für die Methoden einer Ressource keine formalen Parameter definiert werden. Dies ist sicherlich wichtig, wenn man letztendlich Dienste in der Praxis realisiert, aber in dieser Arbeit wird aus Gründen der Übersichtlichkeit von Typisierung abgesehen und es wird stattdessen vielmehr erklärt was einer Methode übergeben wird, also wie der Parameter p beim Aufruf interpretiert wird.

Bei der Bezeichnung der Methode $m \in M_{u,Z}$ stellt man fest, dass es nicht weiter nötig ist, zu annotieren für welche Ressource die Methode gilt. Dies folgt implizit aus der Ressourcenreferenz u und wird im Falle von Unklarheit explizit erwähnt, indem m mit dem Index u annotiert wird (m_u).

Ein *Methodenaufruf* $m(p)$ der Methode $m \in M_{u,Z}$ verursacht außer einer möglichen Zustandsänderung, auch eine (synchrone) Antwort y, die genauso wie die Aufrufparameter p keine weiteren formalen Einschränkungen hat. Wie auch p beim Aufruf der Methode $m(p)$ von der Ressourcenmethode m verstanden werden muss, muss auch im Gegenzug y vom Agenten korrekt verstanden werden.

Ist es von Interesse, die Antwort eines Methodenaufrufs anzugeben, wird das hinter einen Doppelpunkt in der Notation $m(p) : y$ geschrieben. Zusammen mit der Zustandsänderung wird die Notation $m(p)/[Z \rightarrow Z'] : y$ verwendet.

Methoden und Agenten

Wenn es von Interesse ist, dass ein bestimmter Agent A_v eine Methode $m \in M_{u,Z}$ aufruft, dann wird das vor den Methodenbezeichner geschrieben: $A_v : m(p)$. Man wird später erkennen, dass die Annotation des ausführenden Agenten nur dann von Interesse ist, wenn man explizit Aussagen über einen Agenten machen will.

Dies ist oft wichtig, wenn zwei Agenten aus verschiedenen Agentengruppen parallel auf einer Ressource arbeiten und temporalen Zusammenhänge und die Sichtweisen auf die Ressource verdeutlicht werden sollen.

Wird eine Aufruffolge aus der Perspektive der Ressource betrachtet, ergibt sich hingegen eine Abstraktion von dem eigentlichen ausführenden Agenten und damit eine Vereinfachung der Schreibweise. Was schließlich dazu führt, dass es unwichtig ist, welcher Agent eine Ressourcenmethode genutzt hat und wir können diese Notation verzichten. Die Unabhängigkeit des Verhaltens der Ressource von der Identität des Agenten ist auch eine wichtige Eigenschaft, um Caching in dem Architekturstil zu ermöglichen.

Exklusiver Methodenaufruf

Da in einer verteilten Architektur zwei Agenten zu gleicher Zeit jeweils eine Methode auf einer Ressource aufrufen können, muss zuvor geklärt werden, wie man mit diesem Fall ressourcenseitig umgeht. Grundsätzlich gilt, dass Methoden implementierungstechnisch niemals eine Ressource im inkonsistenten Zustand belassen sollten. Natürlich sollte man in der Lage sein zwei Methoden aufrufen, wenn sie ausschließlich lesen. Weil der interne Zustand sich nicht ändert, ist dieser Fall gänzlich unkritisch.

Ändern zwei verschiedene parallele Aufrufe den Zustand Z, sollte es in einer lokalen Transaktion (intern in der Ressource) passieren. Es gilt dabei, dass der erste aufrufende Agent, der eine solche interne Transaktion startet, den anderen, für die Zeit des Request-Reply-Aufrufs, sperren darf. Die Entwickler der Ressource kann bei der Implementierung entscheiden, ob die andere angeforderte Transaktion abgebrochen werden soll oder ob die Transaktion hinter die zu abschließende Transaktion verschoben wird. Im ersten Fall liefert die Ressource bei dem zweiten Methodenaufruf einen Fehler als Antwort und belässt den Zustand wie er ist. Im zweiten Fall kann sich die Antwort verzögern, was zur Folge haben kann, dass das Request-Reply-Paradigma durch Wartezeiten und das Halten von Verbindungen nicht optimal laufen kann. Zudem kann eine Antwort nach der Zustandsänderung verloren gehen, was kritisch für die Konsistenz eines Prozesses sein kann.

Eine alternative Funktionsweise ergibt sich direkt aus modernen Dateisystemen. Ist ein Datenzugriff schreibend, dann erzeugt dieser eine temporäre Kopie der aktuellen Daten auf denen gearbeitet wird. Ist der Datenzugriff beendet, nachdem der Methodenaufruf abgeschlossen wurde, werden die Daten atomar ersetzt. Lesende Methodenaufrufe arbeiten vor dieser Ersetzung auf dem alten Ressourcenzustand. Nach der Änderung wird der neue Ressourcenzustand übernommen und die späteren lesenden Zugriffe sehen auch nur noch diesen neuen Zustand. Dieses Vorgehen hat den Vorteil, dass lesende und schreibende Zugriffe auf die Ressource nicht blockierend sind.

Ressouceninterne *Race Conditions* müssen nicht behandelt werden, wenn zustandsändernde Methoden als eine Einheit gesehen werden. Dies gilt für beide hier aufgeführten Fälle. Race Conditions müssen ressourcenextern behandelt werden, durch Ordnungen auf Ressourcenzugriffen oder durch die Einigung der Agenten auf bestimmtes Verhalten (*Protokoll*).

Interpretationen von Parameter und Antwort

Um der Lesbarkeit Willen, wird in diesem Dokument vereinbart, dass für den Methodenaufruf $m(p) : y$ der Parameter p und die Antwort y ohne weitere formale Einschränkungen konkret angegeben wird. Dazu wird in geschweiften Klammern „$\{\dots\}$“ angegeben, welche Daten genau ein- und ausgegeben werden. Um eine Antwort y einer Methode näher beschreiben zu können, wird eine *Interpretationsabbildung* benutzt. Sie wird mit $\|y\|$ und einem optionalen Index, um mehrere Interpretationen auseinander zu halten.

Zum Beispiel lässt sich eine Methode m_1 aufrufen, die zunächst nur die Antwort y zurückgibt.

$$m_1(\{\}) : y \tag{5.1}$$

Eine vereinbarte Interpretationsabbildung filtert die Werte heraus, die zunächst gekapselt in der Antwort y wurden, damit man übersichtlich und vereinfacht weitere Aufrufe tätigen kann und trotzdem klar bleibt, wo der Wert in der Berechnung entstanden ist. Dies illustriert (5.2).

$$\begin{aligned} &m_1(\{\}) : y \\ &m_2(\{\|y\|_d\}) \end{aligned} \tag{5.2}$$

Die Abbildung $\|\cdot\|_d$ ist hier so vereinbart, dass sie das interessante Datum für die Parameterübergabe an m_2 aus der Antwort y von m_1 herauslöst. Sollte eine sol-

che Interpretationsabbildung gebraucht werden, wird sie explizit erklärt, damit keine Missverständnisse entstehen. Sie dient vielmehr als eine notationelle Hilfe.

In realen Systemen muss für eine saubere Implementierung eine formale Definition solcher Parameter und Rückgaben existieren. Im Falle der Architektur des *WWW*, werden Interpretationen von Repräsentationen für diese Zwecke verwendet. Interpretationen bieten eine Art Mechanismus, um die Antwort für einer Methode auf das individuelle Verständnis des Agenten anzupassen. Im *WWW* kann die Repräsentation durch Priorisierung für die Kommunikation ausgehandelt werden. Weiteres dazu, später im Teil 5.9.

Es ist sinnvoll Interpretationen zu benutzen, wenn die Antwort strukturiert ist, aber man sich mit ihrer Struktur nicht näher befassen möchte, die oft eine komplexe Abbildung ergibt. Genauere Details über komplexe Datentypen würden die Beschreibungen unnötig unübersichtlich machen. Typisierung und Beziehungen auf Typen sind sicherlich eine interessante Facette, wenn man sich mit den Repräsentationen beschäftigt. Der Fokus dieser Arbeit liegt allerdings vielmehr auf der Architektur.

Im Teil 9.1.6 über Simulation wurde die Interpretation verwendet, um aus der Antwort eine mathematische Menge zu konstruieren. So kann man in der Verhaltensbeschreibung (siehe auch Agentenprogramm im Teil 5.3.3) direkt Mengenoperationen benutzen.

5.3 Eigenschaften von Agenten

Spezifiziert man das Verhalten eines Agenten in einem System, reicht es nicht, sich ausschließlich über die Schnittstellen für die Methoden und einzelne Methodenaufrufe Gedanken zu machen. Man muss das Verhalten in einem Kontext betrachten und beobachten, um in einem parallelen System eine Gesamtaussage zu erhalten.

Dazu lassen sich einzelne Methodenaufrufe auf Ressourcen in *Sequenzen* zusammenfassen. Hier wird im Nachfolgenden eine Notation mit Komma benutzt, um den zeitlichen Zusammenhang kennzuzeichnen. Ein wesentliches Merkmal einer verteilten Architektur ist, dass wir keine globale Sicht erhalten können, ohne das System global zu synchronisieren und damit die Parallelisierung zu beseitigen. Aus diesem Grund bedeutet ein Komma zwischen Methodenaufrufen nicht unbedingt, dass keine weiteren Aufrufe im Gesamtsystem auf der gleichen Ressource R passieren könnten. Das Komma ist ein Platzhalter, der semantisch erlaubt, dass Parallelausführungen auf den Ressourcen im System dort andere Methodenaufrufe verursachen können. Um die Auswahl der durch den Platzhalter auszuwählenden Methodenaufrufen im System einzuschränken, werden

die Agenten im System stets alle beschrieben. Auf diese Weise herrscht keine Willkür im System (zum Beispiel Agenten, die zerstörerisch ins System eingreifen), welches gerade beschrieben wird.

Für zwei aufeinanderfolgende Methodenaufrufe $m_1(p_1) \in M_{u_1,Z_1}$ und $m_2(p_2) \in M_{u_2,Z_2}$, wobei $U(R) = \{u_1, u_2\}$, die in Sequenz zu einander stehen, wird die Notation (5.3) benutzt.

$$m_1(p_1), m_2(p_2) \tag{5.3}$$

Dies lässt sich natürlich mit den schon eingeführten Notationen über Methodenaufrufe erweitern. Will man, zum Beispiel, Klarheit über die Zustände der Ressource miteinbeziehen, wird das auf die folgende Weise notiert (5.4).

$$m_1(p_1)/[Z_1 \to Z_1'], m_2(p_2)/[Z_2 \to Z_2'] \tag{5.4}$$

Es ist hierbei klar, dass es entweder mehrere Zustandsübergänge im System vom Zustand Z_1' zum Zustand Z_2 geben muss oder $Z_1' = Z_2$ gilt.

Möchte man hingegen deutlich machen, dass keine Methodenaufrufe an gekoppelten Systemen zwischen den Aufrufen passieren, wird hier das Semikolon gebraucht und auf diese Weise notiert (5.5).

$$m_1(p_1)/[Z_1 \to Z_2]; m_2(p_2)/[Z_2 \to Z_2'] \tag{5.5}$$

In einem System, welches Ressourcen öffentlich zur Verfügung stellt, ist natürlich damit zu rechnen, dass Methodensequenzen beliebig verzahnt werden können. Durch die massive Parallelisierung ist es sogar unwahrscheinlich, dass man einen Aufruf verhindern kann, der ungünstigerweise eine gegebene Sequenz stört. Möchte man Garantien für geschlossene Sequenzen, ist das nur durch externe Steuerung zu erreichen. Diese Steuerungsmechanismen funktionieren zusammen mit Protokollen, deren Eigenschaften im Kapitel 8 näher untersucht werden.

Oft steht, im Falle der Notation mit Semikolon, die lokale Sicht auf einen interessanten Teil des Systems (zum Beispiel das Verhalten einer einzelnen Ressource) im Vordergrund. Oft wird diese Notation dann verwendet, wenn ein konkretes Verhalten gezeigt werden soll für ein mögliches Gegenbeispiel, das gegebene Regeln verletzt.

Lesende Zugriffe auf eine Ressource ändern den internen Zustand zwar nicht, sind aber nicht immer neutral anzusehen. Wegen der Notation mit Komma, kann

eine Verzahnung mit einer zustandsändernden Methode zu unterschiedlichen Resultaten der kompletten Sequenz führen. Der Umgang mit diesen Problemen wird ebenfalls im Kapitel 8 über das Verhalten erklärt.

5.3.1 Agentenreplikation in Agentgruppen

Zwei Agenten A_1 und A_2 gehören zu einer Agentengruppe $A_1, A_2 \in G$, wenn ihr Verhalten, also die resultierenden Methodenaufrufe, im Falle des gleichen ursprünglichen Ressourcenzustands Z, gleich sind.

$$A_1, A_2 \in G \iff \forall Z, m, p : A_1 : m(p)/[Z \to Z'] \text{ und } A_2 : m(p)/[Z \to Z''] \quad \text{mit } Z' = Z'' \tag{5.6}$$

Abhängig von einem Ursprungszustand Z, gibt es hier zwei Fälle für Parallelisierung der Agenten zu unterscheiden:

redundant: die Agenten A_1 und A_2 erkennen den gleichen Zustand Z und führen die gleiche Aktion aus

verteilt: die Agenten A_1 und A_2 entscheiden verschieden anhand von interner Wahl (nichtdeterministisch), was sie unter dem Zustand Z verstehen und vollziehen unterschiedliche Aktionen

Redundanz und Verteilung sind zwei verschiedene Verhaltensarten in Gruppen. Beide Arten von Verhalten können wünschenswert sein und haben verschiedene Ziele in kooperativen Prozessen.

5.3.2 Adressierung von Ressourcen

Da der Agent eine flüchtige Komponente ist, auf deren Existenz und Funktionstüchtigkeit kein Verlass ist, darf dieser kein exklusives Wissen speichern, das für den Fortschritt im Prozess essentiell ist.

Zu solchem Wissen gehören auch *URI*s, mit denen Ressourcen referenziert werden. Zu einem festgelegten Zeitpunkt muss ein Agent auf eine Ressource zugreifen. Und da dieser kein Wissen speichern kann, muss diese Ressource im ganzen System (also für eine bestimmte Anwendung) wohl bekannt sein. Eine solche Ressource, die im System für ihren Zweck systemweit bekannt ist nennt man eine *Wurzelressource*. Mit Hilfe von einer Menge an solchen Wurzelressourcen, muss es dem Agenten möglich sein, durch *Referenzierung* die anderen, für den Prozess relevanten, Ressourcen zu finden. Es gibt ausschließlich diese beiden Mechanismen zur Auffindung von Ressourcen. Entweder ist so eine Ressource eine

Wurzelressource oder sie wird durch die dem Agenten bekanntgemachte Referenzierung gefunden [KN11]. In den Publikationen von Roy Fielding wird für REST in diesem Kontext der Ausdruck „hypertext-driven“ verwendet [Fie08].

In späteren Kapiteln wird nicht mehr darauf explizit hingewiesen, dass die Kriterien für die Referenzierung von Ressourcen erfüllt sein müssen. Es ist in vielen Fällen, die später als Beispiele erwähnt werden, nicht interessant, sich mit dem Thema des „hypertext drive“ zu beschäftigen. Jedoch sollte man für die Praxis beachten, dass die *URI*s nicht einfach im Agenten generiert werden können. Aus Gründen der Komplexitätsreduktion der Verhaltensbeschreibung wurden in vielen Beispielen Quell- und Zielressourcen für Agenten definiert. Diese sind zwar statisch, aber sollten einem Agenten nicht als Wurzelressourcen verstanden werden. Typischerweise würde man den Agenten mit einigen Mitteln ausstatten, um die Ressourcentopologie des Systems untersuchen und die Aufgabe mitteilen zu können, also um welche Ressourcen sich die einzelne Instanz kümmern muss. Im Endeffekt läuft das darauf hinaus, dass in den Systemen noch mehr Ressourcen sind, als die die in den Beispielen erwähnt werden. Da die Topologie dieser Ressourcen aber statisch formuliert werden kann, kann ein Agent diese vollständig lesend erfragen und braucht die Topologie typischerweise auch nicht zu modifizieren.

In einigen späteren Beispielen wurde eine Abbildung $u : R \mapsto U(R)$ vorausgesetzt, die zu einer Ressource eine *URI* ermittelt. Dies ist ein Mittel, um bei den Methodenaufrufen kenntlich zu machen, auf welche Ressource sie sich beziehen, ohne explizit eine neue *URI* einzuführen, die anhand der ressourcentopogischen Beschreibung ermittelt werden kann.

5.3.3 Agentenprogramm

Ein Agent ist die, von der Rechenkraft her, stärkste Komponente in dem hier behandelten Architekturstil. Prinzipiell kann ein Agent in unterschiedlichen Formen existieren. Es wurde bereits besprochen, dass diese abstrakte Komponente auch das menschliche Verhalten formulieren kann.

Beschränkt auf die automatischen Agenten, ist ein Agentenprogramm eine formale Verhaltensbeschreibung bezüglich des kommunikativen Verhaltens zu den angesprochenen Ressourcen. Die hier in der Arbeit verwendete Notation ist nicht vollständig Formal gestaltet, reduziert auf das Wesentliche und dient vor allem zur Erklärung der später eingeführten Eigenschaften im Agentenverhalten.

Die Laufzeit des Agenten, also die Phase, in der der Agent aktiv ist, besteht aus Methodenaufrufen und internen Berechnungen. Ein Agent ist dabei oft so gestaltet, dass sein Programm iterativ und gleichförmig eine Folge (Sequenz) von

Methodenaufrufen durchführt. Typischerweise werden am Anfang des Agentenprogramms Eingaben erfasst. Dabei können Eingaben voneinander logisch abhängen, sodass das Agentenprogramm die strukturelle Referenzierung von Ressourcen nachvollziehen und die Zusammenhänge bereits kennen muss (siehe auch 5.3.2). Auf den gesammelten Eingaben werden Berechnungen durchgeführt und eventuell weitere Eingaben dazu dynamisch hinzugenommen. Abschließend gibt es eine Phase, in der Ausgaben in Zielressourcen gespeichert werden.

Dies ist natürlich der typische Fall, wie ein Agent in einer ressourcenbasierten Architektur operiert. Grundsätzlich ist jedoch ein Agentenprogramm in seiner Form nicht beschränkt. Um in der verteilten Umgebung jedoch eine Kooperation von Agenten und eine konsistente Form der Ressourcenzustände zu gewährleisten, müssen einige Restriktionen bezüglich der Verhaltensweise gemacht werden.

Da die Konsistenz beim Lesen und beim Schreiben von zusammenhängenden, aber verteilten, Informationen gewährleistet werden muss, ist es notwendig, dass das Verhalten einigen Prinzipien folgt, die später im Kapitel 8 weiter untersucht werden.

In dieser Arbeit bestehen Agentenprogramme aus Sequenzen von Methodenaufrufen (die Notation wurde bereits im Teil 5.3 eingeführt), die mit Kontrollflusskonstrukten wie Schleifen oder Entscheidungen angereichert worden sind. In bestimmten Fällen wird `RESTART` benutzt, um festzuhalten, dass die Agenteninstanz die Arbeit zu einem Zeitpunkt in der Zukunft wieder von Anfang an fortführen soll.

5.3.4 Komplexität der Agentenprogramme

Das Agentenprogramm ist, wie schon erwähnt, nicht beschränkt und kann in jeder beliebigen Programmiersprache geschrieben werden, die Kommunikation zwischen Agent und Ressourcen realisieren kann. Wichtig hierbei ist, dass die Komponenten in einer parallelen Multiprozessumgebung funktionieren und damit bezüglich Laufzeit auf einander nicht einwirken.

Für die Beschreibung des Verhaltens von Agenten eignen sich kompilierte und interpretierte Programmiersprachen (auch Skriptsprachen). Beschäftigt man sich insbesondere mit der Ressourcenarchitektur, wie das *World Wide Web*, empfehlen sich Frameworks und Web-Clients, die die Programmierung erleichtern, indem das Protokoll im WWW vereinfacht, durch höherwertige Beschreibungen, benutzt werden kann. So kann sich der Entwickler, unabhängig von den Netzwerkdetails, mit der Anwendung und der eigentlichen Programmlogik beschäftigen.

5.4 Eigenschaften von Ressourcen

Wie bei den Agenten im letzen Teil, lassen sich für Ressourcen in $\mathfrak{R}$, ihre Methoden $m \in M_{u,Z}$ und Aufrufe $m(p)$ ebenfalls einige Besonderheiten innerhalb des Architekturstils feststellen. Diese besonderen Eigenschaften sind nützlich, um das Verhalten von Agenten in $\mathfrak{A}$ und die Auswirkungen auf Zustände der Ressourcen Z einfacher beschreiben zu können.

Geht man von einem dieser hier genannten Merkmale und Prinzipien aus, lassen sich die Konsequenzen für Ressourcen in den Architekturen besser verstehen.

5.4.1 Zustandslosigkeit

Nicht in allen verteilten Systemen ist *Zustandslosigkeit* von zentraler Bedeutung. Zu den hier, in der Arbeit, vorgestellten Paradigmen, das Prinzip der Zustandslosigkeit jedoch wesentlich. Es wird damit vermieden, dass Nebeneffekte erzeugt werden und dies realisiert man, indem man Aktionen in den vorliegenden Architekturen funktional beschreibt.

Ist eine Ressource zustandslos bezüglich der Kommunikation, verhält sie sich exakt gleich, wenn man den Kommunikationspartner austauscht. Konkret heißt das, dass gleiche Methodenaufrufe, wie in (5.7), von der Agenteninstanz selbst unabhängig sind und den gleichen Effekt haben. Alle gleichen Aufrufe mit gleichem Zustand Z veranlassen, dass die Ressource R in den gleichen Nachfolgezustand $Z' = Z''$ übergeht und die Methode $m(p)$, sofern p gleich bleibt, auch gleiche Antwort y liefert.

Sei $m(p)$ ein beliebiger Aufruf einer Methode auf einer Ressource R, die keinen Kommunikationszustand hält. Sei $m \in M_{u,Z}$ mit dem aktuellen Zustand Z und ihrer *URI* $u \in U(R)$, dann gilt wegen der Zustandslosigkeit von R:

$$\begin{aligned} &\forall A_1, A_2 \in \mathfrak{A} : \\ &A_1 : m(p)/[Z \to Z'] : y_1 \text{ und } A_2 : m(p)/[Z \to Z''] : y_2 \implies Z' = Z'' \text{ und } y_1 = y_2 \end{aligned} \tag{5.7}$$

Zustandslosigkeit bezieht sich hier nicht auf den gesamten Ressourcenzustand Z, den man sehr wohl verändern darf (sonst wären Ressourcen nicht so interessant), sondern auf den Zustand bei der Kommunikation mit der Ressource R. Anders gesagt, soll die Ressource R sich beim Aufruf von $m(p)$ gleich verhalten, während ein Zustand Z gehalten wird. Dies gilt natürlich für sowohl für zustandserhaltende, als auch für schreibende Methodenaufrufe, sodass der Effekt der Änderung unabhängig vom aufrufenden Agenten gleich bleibt. Man stellt fest, dass man unter dieser Annahme, auf die Annotation des Agenten (hier: A_1,

A_2) verzichten kann. Im Klartext heißt das, dass eine Ressource R ihren Zustand Z unabhängig von einem aufrufenden Agenten gestalten muss.

Zu beachten ist, dass es bei der besprochenen Zustandslosigkeit und den resultierenden Effekten darum geht, dass es bei $A_1 : m(p)$ und $A_2 : m(p)$ um *Alternativen* geht und nicht um Hintereinanderausführung der gleichen Methode, deswegen sind die Aufrufe in (5.7) nicht mit Komma oder Semikolon getrennt und werden auch nicht in verschiedenen Zuständen aufgerufen. Wäre dies der Fall, würde eine Ausführung von $m(p)/[Z \to Z']$ die Ressource R in den Zustand Z' überführen, was zur Folge hat, dass als nächstes nicht mehr $m(p)/[Z \to Z'']$ aufgerufen werden kann, sondern höchstens $m(p)/[Z' \to Z'']$.

Die Hintereinanderausführung von gleichen Methoden in unterschiedlichen Zuständen ist für den Architekturstil sehr wichtig, um Fortschritt in einem Prozess zu beschreiben und deswegen werden die damit verwandten Eigenschaften als nächstes untersucht.

5.4.2 Instanziierung

Es gibt eine Ausnahmesituation, in der Nebeneffekte erlaubt werden. Dies wird dann zugelassen, wenn Ressourcen instanziiert werden, also bei Neuerstellung einer Ressource R_{neu} von einer anderen Ressource R. Zunächst widerspricht eine Erstellung neuer Ressource den Bedinungen im Teil 5.4.1, weil eine Erstellung der Ressource R_{neu} prinzipiell einem Nebeneffekt in $\mathfrak{R}$ gleicht. Es wird ein neuer initialer Zustand für die erstellte Ressource geschrieben.

Es ist jedoch einfach zu sehen, dass Instanziierung nicht schädlich für ein System ist. Diese Überlegung resultiert daraus, dass laut 5.3.2 R_{neu} keine Wurzelressource ist, die global im System bekannt ist. Das heißt, dass R, als Ersteller, die einzige Ressource ist, die eine Referenz $u_{\text{neu}} \in U(R_{\text{neu}})$ hält.

Somit ist es ausgeschlossen, dass die Nebenwirkung der Erstellung Agenten beeinflussen kann. Sie können keine Änderung im System erfahren, wenn sie nicht zuvor auf die Ressource R zugreifen, welche die Nebenwirkung verursacht hat und einen neuen Zustand signalisiert. Wegen der Bedeutungsneutralität der Referenzen, nimmt man an, dass es ausgeschlossen ist, dass ein Agent u_{neu} einfach errät. Die direkte Adressierung ist nämlich ausschließlich für Wurzelressourcen zugelassen.

Sollte ein Agent A nach der Erstellungsphase temporär eine *URI* u_{neu} erhalten, die von der erstellenden Ressource R nicht abgespeichert wird, dann kann A natürlich ausfallen und die Ressource R_{neu} bleibt im System ohne eine Referenz darauf bestehen. Dieser Fall scheint zwar unerwünscht zu sein, ist aber nicht

schädlich, wenn man an redundantes Verhalten in Agentengruppen denkt. Konstruiert ein anderer Agent A' nämlich eine andere Ressource R'_{neu} mit der er die Teilaufgabe während seiner Aktivitätsphase zu Ende führen kann, ist die tatsächliche Referenz in diesem Fall nicht wichtig, wegen der Bedeutungslosigkeit der *URI*s. Sollte die *URI* jedoch wichtig werden für die Ausführung einer darauffolgenden Aufgabe, dann muss sie abgespeichert werden, sodass sie durch andere Agenten aufgegriffen werden kann.

5.4.3 Idempotenz

Idempotenz ist ein einfacher Mechanismus, um einer sequentiellen Vervielfachung von gleichen Aktionen (oder hier: Methodenaufrufen) die gleiche Auswirkung zu geben wie der einmaligen Ausführung der ersten Aktion. Einige Architekturen werden komplex, um vervielfachte Aktionen geeignet zu behandeln. Manche Architekturen lassen solche Vervielfachungen erst gar nicht zu und ignorieren die Vorteile der Ausnutzung von Redundanz in der Ausführung.

Mit Idempotenz als Werkzeug lässt sich auch für einige einfache Szenarien die Parallelisierung in einer Architektur optimieren, ohne zusätzliche Komplexität einzuführen. Sind mehrere Ergebnisse auf verschiedenen Wegen oder mit unterschiedlicher Effizienz entstanden, kann das Ergebnis verwertet werden, welches am schnellsten verfügbar wurde. Nachfolgende Ergebnisse beeinflussen ein System nämlich nicht mehr.

Wie Zustandslosigkeit (siehe: 5.4.1), ist Idempotenz ein Mittel zur Abstraktion von ausführenden Agenteninstanzen. Hierbei geht es nun vielmehr um die Kontrolle von Aufrufen im Kontext der Gesamtarchitektur. Mit der eingeführten Notation lässt sich Idempotenz folgendermaßen ausdrücken (siehe: 5.8).

$$\begin{aligned}&\text{Methode } m_u \text{ mit } u \in U(R) \text{ ist idempotent, wenn:} \\ &\forall A_1, A_2 \in \mathfrak{A} : A_1 : m_u(p)/[Z \to Z'], A_2 : m_u(p)/[Z' \to Z']\end{aligned} \tag{5.8}$$

Das Komma („,“) in dem Ausdruck deutet darauf hin, dass die Aktionen nicht unmittelbar aufeinander folgen müssen, entsprechend (5.4). Es ist dabei wesentlich, dass der Zustand Z' erhalten bleibt. Das bedeutet, dass lediglich zustandserhaltende Methodenaufrufe zwischen den beiden hier aufgeführten Aufrufen benutzt werden können.

Konkret heißt das, dass wenn die Ressource den Zustand Z' durch einen Aufruf von $m_u(p)$ erreicht hat, bleibt dieser Zustand erhalten, wenn $m_u(p)$ im gleichen Zustand, mit den gleichen Parametern und gleicher Ressourcenreferenz u aufgerufen wird. Jeder wiederholte Methodenaufruf im Zustand Z' verhält sich

ebenfalls zustandserhaltend. Insbesondere bleibt der Zustand der Ressource erhalten, wenn ein Fehler auftritt. Diese Eigenschaft ist anwendbar in Fällen, wenn ein Methodenaufruf nicht gemacht werden kann, weil eine Ressource den Aufruf nicht unterstützt oder formal nicht versteht. Deswegen wird an dieser Stelle $m_u \in M_{u,Z'}$ nicht gefordert und kann dazu gezielt genutzt werden, um Idempotenz zu erzwingen.

Zu bemerken ist außerdem, dass hier explizit A_1 und A_2 benannt werden, obwohl von dieser Unterscheidung eigentlich abstrahiert werden könnte. Insbesondere kann $A_1 = A_2$ gelten.

5.4.4 Safety

Safety ist eine Eigenschaft von Methoden, die aus der HTTP-Spezifikation [F+99] Teil 9.1.1 übernommen wird. Es ist nützlich, diesen Begriff für die ressourcenbasierten Architekturen zu interpretieren.

Eine Methode wird *safe* genannt, wenn sie den gegebenen Zustand Z in $\mathfrak{R}$ unverändert lässt. Insbesondere gilt für die Methode für $m \in M_{u,Z}$, dass der Zustand Z der Ressource R mit Referenz $u \in U(R)$, sich beim Aufruf $m(p)$, mit einem beliebigen Parameter p, nicht ändert (5.9).

$$\forall p : \; m(p)/[Z \to Z] : y \tag{5.9}$$

Die Ressource R und das gesamte System ist nicht in der Lage, festzustellen ob eine Methode m, die safe ist, aufgerufen worden ist, sonst würde das Kriterium $Z \to Z$ nicht zutreffen. So ein Methodenzugriff ist, von der Anwendung aus, nicht messbar[1].

Dieses Verhalten von m gilt vor allem für Methoden, die echt ausschließlich lesend sind und für jede Methode m, die einen Fehler signalisiert. Dieser Zusammenhang wurde bereits in 5.2.2 und 5.4.3 besprochen. Eine Methode, die *safe* ist, ist ebenfalls *idempotent*.

5.4.5 Monotonie

Wie Idempotenz, ist *Monotonie* ebenfalls ein Mechanismus zur Vermeidung von vervielfachten Aktionen oder Methodenaufrufen. Hierbei geht es jedoch vielmehr um verspätete Aktionen, die einen aktuellen Zustand im System auf einen

[1] Netürlich verraten Protokolle und andere Überwachungsmechanismen lesende Zugriffe, aber diese liegen definitionsgemäß außerhalb der Anwendung.

älteren Zustand zurücksetzen könnten und damit den Betrieb in Abläufen stören, beziehungsweise unmöglich machen.

Bei der Berechnung in einem System, welches Informationen speichert und anbietet, möchte man gerne Daten stets aktuell halten können, um einen Fortschritt bei den Berechnungen zu garantieren. Ein nichtverfolgbares Rücksetzen von Daten auf einen alten Stand ist nicht wünschenswert und kann zu diesen Effekten führen:

- allgemeine Inkonsistenz, also fehlerhafte Resultate
- mehrfach durchführte Abläufe, welche von der Idempotenz nicht erkannt werden, was ebenfalls zu fehlerhaften Resultaten führen kann in bestimmten Fällen
- nicht endende Abläufe (wegen *Race Conditions* und *Live Locks*)

Monotonie wird im Folgenden dazu benutzt, um solche Effekte in der vorgestellten Architektur auf passive Weise zu kontrollieren. Monotonie kann, wie Idempotenz, für eine Methode m definiert werden.

Sei t eine Funktion $t : Z \mapsto n, n \in \mathbb{N}$, die die Aktualität eines Zustands Z in R misst. Es wird $\mathbb{N}$ als Wertebereich gewählt, weil eine Totalordnung vorausgesetzt wird zum Vergleichen (somit sind „=“, „<“ und „>“ auf t definiert).

Es wird zunächst nicht betrachtet, wie man die Aktualität aus dem Zustand berechnen kann und wird weiterhin vorausgesetzt, dass anhand der Ordnung auf t entschieden werden kann welcher Zustand Z in R älter ist.

Sei Z der aktuelle Zustand und Z' der Nachfolgezustand, der durch den Methodenaufruf $m(p)/[Z \to Z']$ erreicht wird. Dann gilt (5.10).

$$m \text{ ist } \textit{monoton}, \text{ wenn} \qquad (5.10)$$
$$\forall Z, p : m(p)/[Z \to Z'] \in M_{u,Z} \implies t(Z') \geq t(Z)$$

Die Definition von t erfordert, dass, unabhängig vom Aufrufparameter p, wenn sich der Zustand Z nach dem Aufruf $m(p)$ ändert, dann muss die Ressource entweder einen neueren Zustand gemessen an t einnehmen oder wenigstens in einem gleich alten Zustand verbleiben. Ein Aufruf, der den internen Zustand einer Ressource nicht ändert ist trivialerweise monoton, weil das System keinen Rückschritt in einen älteren Zustand macht.

Diese Form der Monotonie, die im Zentrum der später angestellten Untersuchungen der Ressourceneigenschaften steht. Es gibt noch umfassendere und

strengere Aussagen, die die Untersuchungen noch vereinfachen können. Ist eine Methode so gestaltet, dass sie den internen Zustand Z nicht ändert (ist zum Beispiel ein nichtbeschreibbarer Speicher), dann gilt auch die Monotonieeigenschaft und die Idempotenz gleichermaßen.

5.4.6 Strenge Monotonie

Eine Methode m ist *streng monoton*, wenn jeder Methodenaufruf in dieser Ressource per Definition in den nächsten aktuellen Zustand führt oder die Ressource gar nicht beeinflusst, indem der Methodenaufruf den Zustand unangetastet belässt (siehe: (5.11)).

$$\begin{gathered} m \text{ ist } \textit{streng monoton}, \text{ wenn} \\ \forall Z, p : m(p)/[Z \to Z'] \in M_{u,Z} \implies t(Z') > t(Z) \text{ oder } Z = Z' \end{gathered} \tag{5.11}$$

Strenge Monotonie impliziert Monotonie, da (5.12) gilt.

$$Z = Z' \implies t(Z') \geq t(Z) \tag{5.12}$$

Sowohl Monotonie als auch strenge Monotonie sind ressourceninterne Synchronisierungsmechanismen. Die Funktion t liefert in diesen Fällen eine Ordnung beziehungsweise Totalordnung mit stetig monoton beziehungsweise streng monoton wachsender Aktualität. Die interne Synchronisierung sorgt dafür, dass bei Ressourcen Methodenaufrufe mit älterem Kontext ignoriert werden können. Idempotenz hat eine gewisse Ähnlichkeit zur Monotonie, ist aber eine vollkommen andere Eigenschaft, wie nachfolgend im Teil 5.4.7 gezeigt wird.

In den Ressourcen, die später in Kapitel 7 behandelt werden, stellt sich heraus, dass Monotonie eine Ressource (auf der lesenden Ausgabeseite) mit Eigenschaften ausstattet, die Parallelisierung erlauben. Mehr dazu im Teil 5.6.1. Die strenge Monotonie hingegen führt in Methodenaufrufen dazu, dass redundante Ausführung bevorzugt wird. Auf der Eingabeseite führt strenge Monotonie dazu, dass die Filterung von ankommenden Elementen strikter gestaltet ist.

Ein Beispiel für strenge Monotonie ist eine Ressource, die die aktuelle Zeit in Z speichert. Jeder lesende Aufruf könnte ein $t(Z)$ liefern, welches vom Wert größer ist, als der Vorgänger. Ein anderes Beispiel ist eine Ressource, die eingehende Daten nummeriert. Auch hier entsteht strenge Monotonie auf den Nummern, die ein interner Zähler vergibt.

5.4.7 Untersuchung von Idempotenz und Monotonie

Bevor die Zusammenwirkung zwischen Idempotenz und Monotonie vorgestellt wird, wird zuerst gezeigt, dass sich Idempotenz und Monotonie nicht gegenseitig implizieren und damit vollkommen verschiedene Eigenschaften sind.

Es ist einfach zu zeigen, dass allein idempotente Methodenaufrufe, nicht zur Monotonie führen.

Sei $write(p)$ eine idempotente Operation auf $R \in \mathfrak{R}$ und der Zustand $Z = \{i\}$ umfasst in R eine einzige natürliche Zahl $i \in \mathbb{N}$, die direkt durch Parameter p mit $write$ gesetzt werden kann. Sei t definiert als die Identitätsfunktion $t(\{i\}) = i$.

So ist die Folge der idempotenten Operationen auf Ressource R nicht monoton auf t:

$$write(2), write(1) \tag{5.13}$$

Laut Definition von t speichert das zweite $write$ ein älteres Datum in den Zustand Z. Hiermit wird gezeigt, dass Monotonie nicht durch die Verwendung von idempotenten Methoden zugesichert werden kann. Die Methode $write$ könnte sozusagen ein älteres Datum (gemessen anhand von t) in die Ressource R ungehindert speichern.

Der umgekehrte Fall, dass Monotonie zur Idempotenz führt, ist ebenfalls nicht gültig. Einen kleinen Hinweis dazu gibt es schon im Teil über strenge Monotonie im Teil 5.4.6.

Dazu wird zunächst eine monotone Methode $incr$ definiert, die zum Inkrementieren einer natürlichen Zahl $i \in \mathbb{N}$ im Zustand $Z = \{i\}$ der Ressource R dient. t definieren wir genauso wie im Fall oben als $t(\{i\}) = i$. Da p nicht gebraucht wird, ist der Parameter einfach leer, was im Folgenden mit der Übergabe von „{}“ dargestellt wird.

Man sieht leicht, dass die Folge in (5.14) (streng) monoton auf t ist, weil $t(\{i\}) = i$ nach jedem Aufruf vom Wert größer wird.

$$incr(\{\}), incr(\{\}) \tag{5.14}$$

Jedoch benutzt die Folge selbst nicht idempotente Methoden und damit kann die zusammengesetzte Sequenz ebenfalls nicht idempotent sein. Daraus folgt, dass Idempotenz und Monotonie völlig verschiedene Eigenschaften einer Ressourcenmethode sind, die man getrennt betrachten muss.

5.5 Zusammenwirkung von Idempotenz und strenger Monotonie

Für verteilte Szenarien kann strenge Monotonie unter Berücksichtigung einiger weiterer Eigenschaften für Idempotenz sorgen. Genauere Zusammenhänge werden hierzu im Kapitel 7 vermittelt, wenn Warteschlangentypen besprochen werden.

Im Beispiel 5.14 ist die Monotonie auf einem internen Zustand mittels t modelliert worden. Im Gegensatz dazu zeigt sich, dass t, wenn es einen Bezug zu den übergebenen Methodenparametern hat, ein anderes interessantes Verhalten an den Tag legt.

Bei der Informationsausbreitung ist der Regelfall, dass die Information selbst einen Teil der Methodenparameter ausmacht und die Abbildung t, also die Aktualität, auf diese Information einen Bezug hat. In Informationen in Form von Datensätzen werden oft Metainformationen eingebetet, die Aufschluss über den Ursprung und das Alter der Daten geben. Zusätzlich gilt für äquivalente Daten, die an eine Ressource übergeben werden, dass die Äquivalenz berechenbar ist, wenn eine Ressourcenmethode anhand einer (totalen) Ordnung auf den übergebenen Daten urteilen kann.

Gegeben sei eine Methode $m \in M_{u,Z}$ und zwei Methodenaufe in Sequenz $m(p_1), m(p_2)$ mit Aktualität t, die ausschließlich auf den Parametern ermittelt wird. Es gilt ferner, dass nach dem Aufruf von $m(p)$ für beliebigen Ursprungszustand Z der Zustandsübergang $Z \to Z_p$ ausgeführt wird. Dann kann die Aktualität für den ersten Aufruf von m mit $t(Z_{p_1})$ und für den zweiten Aufruf mit $t(Z_{p_2})$ berechnet werden, jeweils ohne die Berücksichtigung der Zustände vor den Methodenaufrufen. Dann folgt unmittelbar, dass

$$p_1 = p_2 \implies t(Z_{p_1}) = t(Z_{p_2}). \tag{5.15}$$

Ist eine Methode, die t ausschließlich aus Methodenparametern errechnet, streng monoton, dann ist sie ebenfalls idempotent. Es gilt dann nämlich (5.16).

$$t(Z_{p_1}) = t(Z_{p_2}) \implies Z_{p_1} = Z_{p_2} \tag{5.16}$$

Für zwei hintereinanderfolgende Methodenaufrufe bedeutet das (5.17).

$$\begin{aligned} &p_1 = p_2 \implies \\ &(m(p_1)/[Z \to Z_{p_1}], m(p_2)/[Z_{p_1} \to Z_{p_2}] \iff m(p_1)/[Z \to Z_{p_1}], m(p_2)/[Z_{p_1} \to Z_{p_1}]) \end{aligned} \tag{5.17}$$

Damit genügt die Schlussfolgerung der Definition von Idempotenz im Teil 5.4.3.

Diese *Idempotenzkonstruktion* mit Hilfe von strenger Monotonie wird nützlich sein, wenn man sich im Teil 7 mit den monotonen Warteschlangen befasst, die sich, wegen dieser Eigenschaft, hervorragend dazu eignen, verteilte Prozesse zu unterstützen.

5.6 Ein- und Ausgabeverhalten von Ressourcen

Speichert man anhand von Zuständen Z in einer Ressource $R \in \mathfrak{R}$ mehrere Werte in Form von Elementen, dann können diese Elemente als Eingabe für Agenten in $\mathfrak{A}$ dienen. Eine Eingabe für einen Agenten ist gleichzeitig die Ausgabe einer Ressource. Betrachtet man die Ausgabe von R ist diese in erster Linie mit lesenden, zustandserhaltenden Methoden versehen. Möchte man ein echtes Ausgabeverhalten in R haben, müssen Methoden angeboten werden, die gespeicherte Elementwerte monoton oder streng monoton entfernen. Die Ressource R darf beim Entfernen nicht in einen „älteren“ Zustand (bezüglich der Abbildung t) gelangen.

Zur Illustration kann man angeben, dass das Entfernen eines Elements aus einer Menge eine nichtmonotone Operation ist, wenn t die Zahl der Elemente in der Menge berechnet. Der Zustand nach dem Entfernen führt zu einem Zustand der laut t älter ist, da weniger Elemente in der Menge sind.

Da Ressourcen in einem Prozess oft Aufgaben und ihre Reihenfolgen organisieren müssen, ist es wichtig, dass man Elemente entfernen kann. In einem verteilten System will man das Lesen mehrmals erlauben, aber das Entfernen nur ein Mal, wenn eine Aufgabe mit diesem Element abgeschlossen worden ist. Das führt dazu, dass man die Monotonie wiederherstellen sollte, um das Eingabeverhalten nicht zu stören. Trennt man hierbei das Eingabeverhalten vom Ausgabeverhalten einer Ressource, ist man in der Lage Systeme zu entkoppeln, die eine gemeinsame Ressource zur (unidirektionalen) Kommunikation benutzen.

Die Monotonie kann wiederhergestellt werden, wenn anstatt des Löschens ein Mechanismus benutzt wird, der nachvollziehen kann, dass ein Element bereits in der Ressource gespeichert wurde. Mit Hilfe der richtigen Wahl von t lässt sich das im Allgemeinen realisieren. Im schlechtesten Fall bezieht sich t auf die Daten der gespeicherten Elemente und die Ressource braucht so etwas wie einen Möglichkeit zum Ungültigmarkieren der Elemente, die aber noch vorgehalten werden, um die Monotonie auf der Eingangsseite beibehalten zu können.

5.6.1 Redundanz und Parallelisierung

Die Ausgabe der Ressource kann Elemente geordnet (nach einer festgelegten Ordnung) anbieten. Das bedeutet, dass die Berechnungen, die die Agenten auf der Ausgabeseite durchführen, ebenfalls dieser festgelegten Ordnung gehorchen. Die Ressource bietet in diesem Fall eine feste Menge von Elementen an, bis diese als abgearbeitet worden sind, dann wird die nächste Menge anhand der Ordnung bestimmt.

Folgt man diesem Verhaltensschema, ergibt sich ein *redundantes Agentenverhalten* auf der Ausgabeseite der Ressource. Agenten arbeiten zwar verteilt, aber auf gleichen Daten, solange der Ressourcenzustand von der Ausgabeseite her (zum Beispiel durch das Markieren eines Elements als „abgearbeitet"), sich nicht ändert. Redundanz ist hilfreich, weil sie die Wahrscheinlichkeit erhöht, dass eine Aufgabe erledigt wird und zudem eine schnellere Abarbeitung einer bestimmten Aufgabe favorisiert.

Ein anderer Ansatz wäre ebenfalls sehr willkommen in einer verteilten Architektur, der nämlich das *parallele Rechnen* der Agenten auf unterschiedlichen Eingabedaten erlaubt. Um dieses Verhaltensschema zu unterstützen, muss eine Ressource den Agenten, ein zu bearbeitendes Element, frei wählen lassen. Damit wird die Ordnung der Elemente auf der Ausgabeseite der Ressource aufgelöst. Das Ausgabeverhalten kann trotzdem noch als monoton bezeichnet werden, denn das Ungültigmarkieren der Elemente auf der Ausgabeseite der Ressource funktioniert hier und bei der Wahl einer sinnvollen Abbildung t, kann auch der stetige Fortschritt nachvollzogen werden.

Bei dieser Parallisierung gibt es aber auch eine Stolperfalle. Die Ressource selbst ist nicht in der Lage über die Arbeit von Agenten intern eine Information zu speichern. Ein Agent wählt eine Elementenmenge *lesend* und lesende Methodenaufrufe müssen die *Safety*-Eigenschaft erhalten, die im Teil 5.4.4 eingeführt worden ist. Was übrig bleibt, ist den Agenten durch externe Wahl die Elemente bestimmen zu lassen, auf denen dieser arbeiten möchte. Sie können aber nicht ohne weiteres feststellen, ob ein Element gerade von einem anderen Agenten bearbeitet wird.

Die *Redundanz* und die *Parallelisierung* sind zwei gegensätzliche Verhaltensschemata für Agenten. Man kann hier nicht beide Verhalten gleichzeitig im System verlangen, obwohl beide erstrebenswert sein können. Es ist jedoch möglich, durch Kompromisse, das Verhalten zu justieren. Schränkt man zum Beispiel die freie Auswahl der Elemente auf eine Untermenge der gesamt zur Verfügung stehenden Elemente ein, dann verschiebt sich das Verhalten zu Gunsten von Redundanz. In der Praxis ist diese Art von Modifikation bei der Optimierung relevant.

Man kann diese Art von Modifikation einer Ressource mit Hilfe des Prinzips der Paginierung (siehe Teil 7.9.1) erzeugen.

5.7 Komponentenarchitektur

Spätere Kapitel behandeln Themen rund um den Transport von Daten und Informationen, und die sich daraus ergebenden Eigenschaften für die Bildung von Protokollen, Sperrmechanismen und Synchronisation. Es ist deswegen wichtig die Architektur aus der Perspektive des Software-Engineerings vorzustellen, anhand welcher diese Eigenschaften Vorteile bringen.

Im Hintergrund der Architektur steht das Prinzip von Entkopplung der Aktivitäten eines Prozesses. Die Abwesenheit von aktiven Komponenten soll den Prozessverlauf möglichst wenig beeinflussen und stets so weit vorantreiben, wie es zum aktuellen Zeitpunkt noch möglich ist. Bei der Wiederkehr der vorher abwesenden aktiven Komponenten, soll das System geeignet weiter fortschreiten.

Die erste Feststellung war, dass Daten und aktive Komponenten der Architektur getrennt werden sollten. Der Grund ist, dass Daten viel einfacher durch technische Mittel redundant gehalten werden können als Rechenkraft.

Zusätzlich wird gefordert, dass die aktiven Komponenten in der Architektur keinen privaten Zustand halten dürfen, welcher eine Berechnung im Systems beeinflussen könnte, wenn die Berechnung aus irgendwelchen Gründen stoppt. Das Prinzip der *Zustandslosigkeit* 5.4.1 ist hier essentiell.

Daraus ergibt sich ein System, welches auf Agenten und Ressourcen basiert. Die Suche nach einem Architekturstil, welches obige Eigenschaften unterstützt, führt vor allem zu Multiagentensystemen [SLB09]. Man stellt aber schnell fest, dass Agenten in Multiagentensystemen einen Status halten und die Trennung zwischen Daten und Aktionen nicht passend für die genannten Ausfallszenarien von Agenten organisiert ist. Agenten eines Multiagentensystems sind auch nicht heterogen. In einem Unternehmensprozess ist es durchaus üblich, dass Agenten verschiedene Aufgabenbereiche in einem parallelen System haben. Prozesse so zu entwickeln, dass jeder Agent jedes Aufgabenfeld beherrscht, ist meistens (bis auf Sonderfälle) nicht erwünscht. Hier unterscheidet sich der Architekturstil von Multiagentensystemen prinzipiell.

Eine passendere Architektur bietet das *World Wide Web*, welches auf den Prinzipien des Architekturstils *REST* basiert. Diese Architektur wird nun näher untersucht und in Verbindung mit den zuvor angestellten Überlegungen gebracht. Das *WWW* und *REST* lassen das Verhalten der Agenten in der Architektur voll-

ständig offen und spezifizieren technische Aspekte der Ressourcen, die im nächsten Teil 5.7.1 besprochen werden.

5.7.1 Bezug zu REST

Im Jahr 2000 ist eine Arbeit von Roy Fielding mit dem Titel „Architectural Styles and the Design of Network-based Software Architectures“ [Fie00] erschienen, die die Philosopie hinter dem WWW (world wide web) näher erklärt hat. Diese Arbeit machte den Architekturstil namens „REST“ (*representational state transfer*) bekannt, der zur Zeit einen kleinen Wirbel in der Web-Services-Gemeinde erzeugt hat. Während der *WWW conference* [WWW12] werden auch jedes Jahr Workshops zum Thema „RESTful Design“ durchgeführt [WSR12].

Zu beachten ist, dass REST ein Architekturstil ist, also vielmehr ein Paradigma nach dem Architekturen aufgebaut sind. Hier wird nicht näher auf REST allgemein Bezug genommen, sondern auf die Architektur des Web, die als Protokoll HTTP [F+99] benutzt und welches von REST maßgeblich mitgeformt wurde. Das Akronym „REST“ wird hier oft im Kontext von *WWW* gebraucht, ohne darauf explizit aufmerksam zu machen, weil mit dem *WWW* der praktische Bezug besser verstanden werden kann. Der Architekturstil „REST“ ist jedoch allgemeiner.

Was REST interessant macht, ist die Nutzung der Idempotenz bei den *Protokollverben*, die durch das verwendete Protokoll definiert werden. Es ist eine Eigenschaft, die erleichtert, die Kommunikation selbst funktional (das heißt: zustandslos) zu betrachten. Dadurch ist es leicht, in das *WWW Caching* einzubauen, was die Architektur von der Effizienzseite auf einfache Art und Weise stärkt. *Caching* ist hierbei nicht zu unterschätzen und sollte bei der Entwicklung von Web-Anwendungen stets berücksichtigt werden. *HTTP* wurde darauf explizit hin entwickelt, die Caching-Hierarchien zu unterstützen. Auf diese Weise wird Last von Diensten genommen, die insbesondere agnostisch gegenüber der Identität der Client-Komponenten sind (in ressourcenbasierten Architekturen sind Agenten die „Clients“ für Ressourcen).

Diese vielen strukturellen Gemeinsamkeiten bezüglich der Anatomie und den Prinzipien rund um die Web-Dienste (oder auch Ressourcen) führten dazu, dass REST dazu verwendet werden kann, die ressourcenbasierte Architektur, die hier erörtert wird, praktisch zu erforschen. Es ist jedoch wichtig anzumerken, dass REST lediglich als eine bestimmte Instanz von ressourcenbasierten Architekturstilen behandelt wird, die sehr naheliegend ist und zudem praktisch relevant ist. Mit etwas mehr Aufwand ist, es möglich, mit jeder Sprache, die Kommunikation zwischen Prozessen und datenbasierten Diensten erlaubt, die Merkmale aus dem Teil 5.1 nachzubilden.

Eine andere Ausprägung ist, zum Beispiel, die Programmiersprache *Go* [GoL12], die ganz unabhängig von REST für die Experimente dieser Arbeit benutzt worden ist. Die dort eingesetzten Ideen aus der bekannten Prozessalgebra „Communicating Sequential Processes“ (CSP) [Hoa04] [Hoa78], lösten das Problem der Kommunikation, indem die „Inter Process Communication“ (IPC) [Ste98] sehr einfach gestaltet werden konnte. Man kann ebenfalls eine Ressource als einen kommunizierenden Prozess verstehen, der Methoden in Form von RPC („remote procedure call“) zur Zustandsänderung anbietet. Die Experimente mit der Alternative *Go* wurden ausschließlich mit dem Konzept des IPC auf einem einzelnen Rechner durchgeführt. Es gibt aber auch API-Erweiterungen für *Go*, die erlauben CSP in Form von *network channels* rechnerübergreifend und völlig transparent anzubieten, sodass es völlig ausreicht Experimente lokal zu betrachten.

Im weiteren Verlauf wird jedoch vor allem von REST und HTTP, als Grundlage für Experimente und Erklärungen, Gebrauch gemacht, deswegen wird hier als nächstes untersucht, wie sich die Eigenschaften von REST und der etwas allgemeiner gestalteten Ideen aus Teil 5.1 zu einander verhalten.

5.7.2 Ressourcen

Eine Ressource ist eine Komponente, in der *Information* gespeichert wird. Zuvor wurde diese Information mit dem Ressourcenzustand Z bezeichnet. Auf eine Ressource kann man mit *Methoden* zugreifen, die laut Ressourcenzustand aktuell verfügbar sind (siehe auch: Teil 5.2.2).

Ressourcen können miteinander eigenständig nicht kommunizieren. *Informationsfluss* zwischen Ressourcen wird mit Hilfe von Agenten (siehe nachfolgende Definition in 5.7.3) realisiert.

Weiterhin werden Ressourcen mittels *URI*s angesprochen, die für dynamisch erstellte Ressourcen bedeutungsneutral sind und insbesondere keine Kodierungsschemata aufweisen, die Semantik in die Struktur bringen. Auf der anderen Seite stehen Wurzel-Ressourcen, die im *WWW* wohlbekannt sind und einen Einstiegspunkt darstellen, mit welchem sich der Weg zu anderen, in dieser Architektur verstreuten, Informationen finden lässt.

5.7.3 Agenten

Ein Agent ist eine Komponente, die in der Lage ist, Methoden auf Ressourcen aufzurufen, um Information zu sammeln, zu verarbeiten und in Ressourcen zu speichern. Im Architekturstil REST wird der Agent oft als „Client“ bezeichnet,

weil er das Gegenstück zum Web-*Server* bildet, der Ressourcen zur Nutzung exponiert.

Agenten können miteinander eigenständig nicht kommunizieren. Agentenübergreifender *Kontrollfluss* wird mit Hilfe von Ressourcen realisiert. Architekturtechnisch ist ein Agent ein Stück Software, die automatisiert oder manuell gesteuert werden kann. So können bestimmte Programme oder Skripte Ressourcenmethoden aufrufen und bestimmte Effekte in Form von Zustandübergängen verursachen.

Beispielweise ist es denkbar, dass ein Mensch hinter einem Web-Browser („Browser Agent“) Ressoucenmethoden aufruft. Ein Web-Browser ist durch bestimmte Respräsentationen multipler Ressourcenzustände in der Lage visuelle Angepassungen zu machen, um komplexe Sachverhalte dem Menschen als Benutzer, kompakt und in der richtigen Form, zu präsentieren. Des Weiteren sind Methodenaufrufe von der Technologie des HTTP-Protokolls abstrahiert und werden semantisch vom Benutzer gedeutet. Dies geschieht in Form von Repräsentationen, die von Ressourcen ausgeliefert werden. Damit ein Mensch einen verständlichen Überblick über den Ressourcenzustand hat und die resultierende Anfrage an die Ressource interpretieren kann, kommt oft Hypertext in Form von verschiedenen Markup-Sprachen zum Einsatz.

5.8 Protokollprimitive für REST

Technische Dokumente, die sich mit REST tiefer befassen, beschreiben die vier bekannten Verben zur Kommunikation mit einer Ressource. Ein Protokoll nach dem REST-Prinzip versucht *Methoden* $m \in \mathfrak{M}$ auf *Verben* des Protokolls HTTP [F+99] so abzubilden, dass sie dem fest definierten Charakter der Methode entsprechen.

Mit der Benutzung von $\mathfrak{M}$ wurde im Teil 5.2 direkt eine Abstraktion von technischen Protokollen gemacht. Da ohne diese Protokollprimitive, die REST und HTTP erfordern, konkrete Experimente in diesem Architekturstil nicht durchführbar wären, werden sie hier vollständigkeitshalber erwähnt.

Die Verben sind alle gebunden an eine Referenz u und entsprechen grundsätzlich einer Methode $m \in M_{u,Z}$, die mit der *URI* u und dem Zustand Z definiert ist. Das u wird hier direkt an das Verb annotiert, weil in REST die Referenz u ein wichtiger Bestandteil des Protokolls ist, der unabhängig von p ist, und auf den in diesem speziellen Architekturstil Bezug genommen werden muss.

Anders gesagt, kann man den Verben erst eine semantische Bedeutung eines Methodennamens in $\mathfrak{M}$ geben, wenn die Referenz u bekannt ist, auf welcher die Me-

thode definiert ist. Die exakte Bedeutung eines Aufrufs ist allerdings auch von Z abhängig. Zur Erklärung der charakteristischen Eigenschaften der Verben, ist der Zustand der Ressource allerdings nicht nötig.

$GET_u(p)$: Lese eine Repräsentation der Untermenge des Ressourcenzustands Z über die Referenz u unter Berücksichtigung des Parameters p.

$PUT_u(p)$: Beschreibe mittels Referenz u und dem übergebenen Parameter p eine Untermenge des Ressourcenzustands Z.

$DELETE_u(p)$: Lösche Referenz u. Zusätzliche Informationen zur Löschanforderung sind im Parameter p.

$POST_u(p)$: Führe mittels Ressource R mit zugehörigem Zustand Z hinter der Referenz u eine mit p parametrisierte Operation aus, die nicht auf GET, PUT oder $DELETE$ passt und Z modifizieren und eventuell neue Ressourcen in $\mathfrak{R}$ erzeugen kann.

Um eine Methode passend auf ein solches Verb abzubilden, müssen die Eigenschaften der Methode analysiert werden. Jedes Verb hat eine charakteristische Funktionsweise. Die Verben weisen eine gewissen Ähnlichkeit zu „Create, Read, Update and Delete“ (C.R.U.D.) auf [Mar83], jedoch ist $POST$ ein viel allgemeineres Verb und entspricht nicht nur dem „Create“.

5.8.1 GET

Das Protokollverb *GET* wird verwendet, wenn der Zustand Z der Ressource R abgefragt werden soll und Z dabei nicht verändert wird. Es soll also laut Z nicht feststellbar sein, dass ein beliebiges $GET_u(p)$ jemals aufgerufen worden ist. Da dieses Verb keine Zustandsänderungen verursacht, wird es in der Spezifikation im Gegensatz zu den anderen drei erwähnten als „safe“ bezeichnet ([F+99] 9.1.1), woraus auch die Definition der Safety in 5.4.4 abgeleitet worden ist.

In der Praxis findet man Web-Dienste, die nicht seiteneffektfrei mit *GET* umgehen. Dies wird hier für die behandelten Ressourcen nicht erlaubt. Die HTTP-Spezifikation [F+99] wird an dieser Stelle streng befolgt.

Die Funktionalität von *GET* auf diese Weise zu beschränken ist wichtig, weil der Sinn dieses Verbs ist, dass Caching-Mechanismen innerhalb einer Software-Architektur unterstützt werden können. Caching [Fie00] ist ein wichtiges Prinzip zur Entlastung von Ressourcen im Web.

GET ist zwar, von den Möglichkeiten her, das schwächste Verb (modifiziert eine Ressource nicht), aber auch das am meisten verbreitete in HTTP. Es ist das ein-

zige Verb, welches in der Lage ist, Daten in Caches zu laden und es ist auch das einzige Verb, welches von Caches profitiert.

5.8.2 PUT

PUT ist ein Verb, welches das Beschreiben von einer Untermenge von Z in einer Ressource R erlaubt. Es ist darauf zu achten welche Art von Schreiben hier gemeint ist. Im Vordergrund steht hier vor allem das *idempotente* Beschreiben von Teilen des Ressourcenzustands. Dies ist ein notwendiges Kriterium für die Wahl von *PUT*. Ein weiteres Kriterium ist, dass *PUT* von der Bedeutung her einer *Zuweisung* von Daten zu Z entspricht. Die Zuweisung ist dabei fest gebunden an die Referenz $u \in U(R)$ und darf außerhalb von Z keine Modifikationen in der Welt $\mathfrak{R}$ durchführen. Dies ist ein wichtiges Kriterium zur Unterscheidung zwischen *PUT* und *POST*.

Dies bedeutet, dass PUT zwar für den noch nicht definierten Zustand Z aufgerufen werden kann (also, wenn die Ressource R noch gar nicht existiert) und damit die Ressource erzeugen darf, aber dabei dürfen keine anderen Ressourcen erzeugt werden. Diese Ressourcen würden sonst andere $u_{\text{neu}} \notin R$ und Z_{neu} haben und verursachen Seiteneffekte, die für *PUT* nicht erlaubt sind.

Die Kontraindikation für *PUT* ist damit gegeben, wenn außer innerhalb von Z andere Ressourcenzustände erzeugt oder modifiziert werden. Oft sind diese Aktionen bereits nicht idempotent, wie, zum Beispiel, das Anhängen eines Elements an eine Listenstruktur oder eine Methode, die neue Referenzen für die Ressourcen erzeugt.

Man muss hier aber auch etwas vorsichtig sein, denn auch idempotente Methoden können neue Ressourcen erzeugen, indem der übergebene Parameter die Erzeugung steuert. Im obigen Beispiel kann Idempotenz konstruiert werden, indem man das mehrfache Erzeugen von Ressourcen mit gleichem Inhalt ignoriert. Laut den oben genannten Kriterien wird jedoch immer noch, beim ersten Aufruf, eine neue Referenz auf die konstruierte Ressourcen gebraucht, was einen für PUT unzulässigen Seiteneffekt zur Folge hat.

Allerdings spricht nichts dagegen, beispielsweise, den Zustand der einzelnen Elemente einer Liste mit *PUT* zu modifizieren, wenn es nicht erforderlich ist, dass die überliegende Listenverwaltung diese Modifikationen berücksichtigen muss. Dies ist zum Beispiel der Fall, wenn die Listenverwaltung lediglich *URI*s zu den eigentlichen Elementen speichert.

5.8.3 DELETE

Ein Aufruf von $DELETE_u(p)$ verhält sich anders als man es von der Beschreibung von *PUT* vermuten würde. Es modifiziert nämlich nicht Teile des Ressourcenzustands Z, sondern löscht die Referenz u, sodass u in der Zukunft nicht mehr gültig ist. Die Ressource R verliert die Assoziation mit dem Zustand Z aus der Perspektive von $u \in U(R)$. Es ist zu beachten, dass andere $u' \neq u$, mit $u' = U(R)$, mit dem Zustand Z assoziiert sein können und diesen unbeeinflusst lassen müssen.

Das Verb hat ebenfalls einen idempotenten Charakter in Bezug auf die Referenz u. Die Referenz wird beim ersten Aufruf invalidiert. Mehrfache aufeinander folgende Aufrufe von $DELETE$ auf dieser Referenz haben keinen weiteren Effekt.

5.8.4 POST

Die Abbildung einer Methode auf das Verb $POST$ wird dann benutzt, wenn eine erweiterte Funktionalität bezüglich der Ressource R verlangt wird. Zunächst ist *POST* die einzige Methode, die nicht zwingend idempotent sein muss. Es wird also erwartet, dass so ein Aufruf Nebenwirkungen haben könnte, die nicht in Z als Zustand von R abgebildet werden müssen.

Anders als bei $PUT_u(p)$, beeinflusst $POST_u(p)$ nicht unbedingt die zu Ressource R mit der Referenz u, sondern ist in der Lage weitere Ressourcen zu instanziieren (erstellen).

POST kann also als ein Platzhalter benutzt werden für den keine anderen Verben zutreffen. Es kann sogar die anderen Methoden ersetzen, was einige Implementationen von Web-Applikationen in der Praxis tatsächlich machen. Die Einschränkung auf das Verb *POST* gilt allerdings als schlechter Stil bei REST, schon alleine deswegen, weil Cache-Hierarchien am besten unterstützt werden, wenn *GET* richtig genutzt wird.

$POST$ kann ebenfalls den Zustand einer anderen Ressource modifizieren. Dies ist jedoch nicht immer gute Idee, denn solche Art von Methodenaufrufen die Konsistenz in Caches beeinflusst, weil der Cache eine solche Änderung nicht direkt mitbekommt. Eine Ressource, die durch Nebenwirkungen beeinflusst wird, muss in diesem Fall durch Metadaten entsprechend beschrieben worden sein, sodass ein Agent diese Art von relaxierter Konsistenz erwartet. Methoden, die Nebeneffekte verursachen, werden allerdings in dieser Arbeit vermieden.

5.9 Repräsentationen

Bei HTTP [F+99] im REST-Architekturstil [Fie00] haben Repräsentationen einen besonderen Stellenwert. Um bestimmte Arten von vordefinierten Repräsentationen zu erkennen wurde eine Norm mit dem Namen *MIME* in Form eines RFC[2] (RFC-2046: Media Types [FB96]) verfasst, woraufhin viele andere Erweiterungen dieses RFC entstanden sind, das ursprünglich für die Übertragung von E-Mails gedacht war. Dieser akzeptierte Standard floss auch in den HTTP RFC ein und eröffnete die Möglichkeit, *Dokumententypen* in der Kommunikation beim Web-Protokoll zu verwenden.

Im Teil 5.2.2 wurden Repräsentationen nicht weiter betrachtet. Der dort spezifizierte Methodenaufruf $m(p) : y$ erlaubt die freie Gestaltung der Anfrageparameter p und der Antwort y. Es ist im eigentlichen Sinne eine Abstraktion von diesen speziellen Eigenschaften, die HTTP nutzt. Im Gegensatz zu den theoretischen Betrachtungen, die Kontrolle und Informationsfluss erörtern, sind bei HTTP nach dem REST-Architekturstil Repräsentationen wichtig. Eine der Prämissen von REST ist nämlich, dass man die Kontrolle der gesamten Applikation (also des Zusammenspiels von Agenten und Ressourcen) über Repräsentationen abhandeln soll [Fie00]. Das bedeutet, dass man kein *implizites Wissen* haben sollte, was als nächstes geschehen soll, sondern, dass sich die Applikation von selbst beschreibt und die Möglichkeiten zur Fortführung stets anbietet. Der Sinn dahinter ist, dass auf diese Weise Ressourcen einen Applikationszustand widerspiegeln und mit der Antwort y gleichzeitig alle Informationen zur Fortführung eines Prozesses mitliefern. Ein anderer Grund ist, dass Agenten kein implizites Wissen vorhalten müssen, welche Möglichkeiten für den nächsten lokalen Schritt existieren. Die Gesamtanwendung beschreibt also zu jedem Zeitpunkt die Möglichkeiten für ihren Fortschritt selbst. Durch diese Art von Verbindungen mittels Referenzen von und zu Ressourcen bekam das *WWW* als „verwobene“ Architektur ihren Namen.

Da das WWW im Vordergrund für Menschen entwickelt worden ist, ist der Aspekt der Steuerung durch Repräsentationen äußerst wichtig und insbesondere in Verbindung mit dem Thema Crowdsourcing, da hier Menschen als Agenten die verteilte Rechenkraft darstellen. Des Weiteren ist, allgemeiner, überall wo der Mensch ein Teil des Prozesses ist, an diese Empfehlung zu denken, denn Menschen brauchen als Agenten Anleitung über die Möglichkeiten, die gerade zur Verfügung stehen. Die Denkprozesse und die Arbeitsweise der Menschen in einem System lässt sich äußerst schlecht formalisieren, aber, durch Repräsentationen, etwas in die beabsichtigte Richtung steuern.

[2] engl.: request for comment

Für das entwickelte Framework für Experimente auf Basis von REST, war es wichtig, Repräsentationen genauer zu betrachten. Sie schaffen die Möglichkeit, in Anfragen und Antworten Standards zu nutzen, was wiederum die Entwicklung von Agenten durch vorgefertigte Interpretationsmechanismen stützt (zum Beispiel Bibliotheken und APIs, die bestimmte Daten-/Dateiformate verstehen).

Ein *Methodenaufruf* $m(p)$ liefert eine durch p wählbare Repräsentation $\|\|\|_p : y \mapsto \|y\|_p$ einer *(synchronen) Antwort* y (siehe Definition in 5.2.2). Diese Form von Definition ist allgemein akzeptabel, aber, für Repräsentationen nach REST, nicht präzise genug.

1. In Praxis ist eher nie die gesamte Spezifikation von p für die Wahl der Repräsentation zuständig.
2. Vom Architekturstil her will man vermeiden, dass die Wahl der Repräsentation ein von der Semantik her anderes y liefert.
3. Das gleiche gilt für den Ressourcenzustand Z. Die Wahl der Repräsentation sollte den Zustand nicht beeinflussen.

In praktischen Implementierungen würde man also die Anfrageparameter unterteilen:

p_b**:** Körper der Anfrage, der sich auf Z auswirken kann.

p_h**:** Kopf der Anfrage, der Steuerungsinformationen bezüglich der Anfrage in p_b enthält, die sich nicht direkt auf Z auswirken, sondern lediglich spezifizieren, wie die Anfrage übermittelt wird.

Die gleiche Unterteilung ist im Falle von HTTP bei der Antwort y mit y_h (sogenannter *Header*) für Steuerungs- und Metainformationen und y_b (*Body*), dem eigentlichen Inhalt der Antwort. Die Steuerungsinformationen mögen spezifische Angaben sein, die die technische Architektur im verteilten System unterstützen (zum Beispiel Angaben über die Gültigkeit der Antwort, um Caching zu unterstützen). Metainformationen sind Informationen über die Form des Inhalts y_b. Das mag auf einer Seite der Dokumententyp sein, den der schon am Anfang angesprochen wurde, aber auch Angaben zur Übermittlung und Kodierung von y_b.

5.10 REST oder nicht REST?

Nicht alles was das HTTP benutzt (oder sogar auch in der Praxis „REST“ genannt wird) ist sinngemäß nach der Idee von REST entwickelt worden. Es wurde schon

erwähnt, dass einige Web-Diensteanbieter das HTTP-Protokoll etwas weit auslegen und für zu viele Methodenaufrufe generell das Verb *POST* einsetzen. Damit umgehen sie einige technische Unzulänglichkeiten von Web-Agenten (Browsern, die teilweise kein PUT und DELETE beherrschen), aber ignorieren einen Teil des Protokollverhaltens, welcher wichtig für REST ist, nämlich die Cache-Architektur. Im Extremfall gibt es sogar Web-Dienste, die ausschließlich *POST* benutzen und damit jeglichen Vorteil von REST verspielen.

Am ursprünglichen HTTP-Protokoll, welches den REST-Prinzipien entstammt, wurde eine inzwischen weit akzeptierte Veränderung außerhalb der Standardisierung der Internet Engineering Task Force (*IETF* [IET12]) gemacht. Es handelt sich um sogenannte *Cookies*, die nicht konform sind mit dem Prinzip der Zustandshaltung der Applikation. Roy Fielding hat viele Male gewarnt, dass Cookies, die weit verbreitet sind, um HTTP-Sessions nachzuverfolgen, die *Zustandslosigkeit* der Ressourcen zerstören, die im Teil 5.4.1 vorgestellt worden ist [Fie00]. In dieser Arbeit werden solche Mechanismen wie Cookies, aus diesem Grund, nicht benutzt.

Kapitel 6

Test-Framework „cgi2c“

Die ressourcenbasierte Architektur aus dem vorhergehenden Kapitel 5 gibt Anlass ein Werkzeug zu entwickeln, welches mit möglichst wenig Aufwand, die Entwicklung von datenstrukturgetriebenen Ressourcen ermöglicht, deren Implementation wiederverwendbar ist.

Dass REST-Frameworks diesbezüglich sehr rudimentär ausgestattet sind, zeigt der Teil 6.1. Dies wurde als Motivation genommen, eine Experimentallösung zu erstellen, mit Hilfe welcher die Umsetzbarkeit und Funktionsweise weiter erforscht werden konnte. Der Projektname *cgi2c* deutet darauf hin, dass die zu erstellende Middleware eine Umsetzung der standardisierten CGI-Schnittstelle [RCF04] auf C-Code machen soll. CGI ist eine sehr einfache Anbindung, die von Webservern benutzt wird, um Kommunikation zwischen Clients und der Anwendung hinter der Web-Server-Komponente zu ermöglichen.

Bevor die Realisierung im Detail vorgestellt wird, werden einige Überlegungen aufgeführt, die zum Design beigetragen haben. Es ist sicherlich interessant diese zu erwähnen, weil diese Überlegungen einige komplexe Zusammenhänge im Entwurf aufzeigen, die bei naiver Herangehensweise gar nicht aufgefallen wären und zum Scheitern der Erstellung einer funktionstüchtigen Software-Komponente geführt hätten.

Bei der Untersuchung von REST-konformen Applikationen, ist zunächst festzustellen, dass man sich nicht einig ist, was REST eigentlich bedeutet. In der Realität wurden bereits Web-Applikationen, die behauptet haben, „RESTful“ zu sein, unter den Kennern als nichtkonform bezeichnet. Das liegt daran, dass ein Entwickler-Framework keine Möglichkeiten hat, die eigentliche Implementierung gegen dieses Fehlverhalten zu schützen. Es ist zum Beispiel nicht weiter möglich, zuzusichern, dass ein Diensteentwickler beim Verb GET einen internen Zustand modifiziert, was sofort zur Verletzung des Safety-Kriteriums für dieses Protokollverb führt. Anderes typisches Problem ist die Misachtung der

Bedeutungsneutralität von URIs (englisch: „opaqueness“), indem unter Umständen URIs nach einem bestimmten Schema rekonstruiert werden können, ohne dass es aus der Applikation hervorgeht und der Antrieb mittels Repräsentationen hier nicht korrekt funktioniert.

Es ist vielmehr so, dass der Entwickler eines Web-Dienstes dabei unterstützt wird, den Aufwand bei der Entwicklung und der Integration eines solchen Dienstes von der technischen Seite her klein zu halten.

6.1 Existierende Frameworks und Middleware für REST

Es gibt mindestens drei bekannte Java-basierte Entwicklungs-Frameworks zur Anbindung von Web-Diensten per REST.

- Jersey [Jer12]
- Restlet [Res12b]
- Restfulie [Res12a]

Jersey und Restlet sind REST-Frameworks, die für die Programmiersprache Java [Jav12] geschrieben worden sind. Restfulie kommt in drei Varianten von Programmiersprachen: für Java, C# und Ruby. Die Aufgabe aller dieser Frameworks ist, die Entwicklung von REST-Diensten, für die Server- und die Client-Seite einer Diensteanwendung, zu unterstützen.

Analog zu Servlets [PL02], sind REST-Dienste ebenfalls in Klassen gekapselt und werden an feste URIs im HTTP-Server gebunden. Im Gegensatz zu Servlets sind REST-Dienste leichtgewichtiger (Möglichkeit auf Verzicht auf die Servlet-Container-Komponente) und in der Lage protokollspezifische und anwendungsspezifische Aspekte voneinander zu trennen.

Alle diese REST-Entwickler-Frameworks funktionieren so, dass sie HTTP-Methoden auf die Methoden in der jeweiligen unterstützten Programmiersprache delegieren. Dazu wird eine Ressource als Klasse oder direkt deren Methoden mit dem entsprechendem HTTP-Verb und mit einer URI assoziiert. Die einzelnen Methoden haben, während der Abarbeitung der HTTP-Anfrage, die Möglichkeit, auf zusätzliche Protokollinformationen zuzugreifen, die im Anfragekopf, Anfragekörper und als Aufrufparameter übergeben worden sind. Diese Informationen sind vorverarbeitet, sodass sie im Kontext der jeweiligen Programmiersprache einfacher und logischer verfügbar sind.

Zusätzlich, gibt es in der protokollspezifischen Behandlung von REST, in allen diesen Frameworks, eine rudimentäre Unterstützung für Repräsentationen. Das

bedeutet, dass mit einer Methode der zugehörige Antwort-MIME-Typ (Datentyp der Antwort) assoziiert werden kann. Die anwendungsspezifische Seite von REST wird so unterstützt, dass in die Frameworks einige für REST gebräuchliche Repräsentationen, wie zum Beispiel XML [W3C12b] oder JSON [JSO12], während der Objektserialisierung, erzeugt werden können. Etwaige dynamische Inhalte der Antwort werden durch die Repräsentationen in der Art unterstützt, dass die zugehörige (native oder externe) Programmierschnittstelle des abstrakten Datentyps verwendet wird. Zum Beispiel ist dies im Falle von XML, die DOM-API [W3C05], die typisch dafür ist, aber keine direkte Applikationssemantik und zugehörige Zustandsverwaltung behandelt, was sich bei späteren Untersuchungen als durchaus komplex herausstellt.

6.2 Umsetzung auf Web und REST

Erstens ist die Frage zu klären, warum das *WWW* als Architektur gewählt worden ist. Ein Experimental-Framework kann sehr gut auf anderen Architekturen mit anderen Architekturstilen neben REST sehr gut konstuiert werden.

REST hatte einen Einfluss auf die Arbeit, weil es an vielen Stellen auf Komplexität in der Anwendungsentwicklung verzichtet. Im Endeffekt lässt sich durch ein REST-Framework eine leichtgewichtige Bereitstellung von Daten in typisierter und standardisierter Form realisieren. Das hat den Vorteil, dass durch den HTTP-Standard viele einfache Werzeuge (HTTP-Clients in vielen verschiedenen Formen) zum Einsatz kommen können. Das reduziert den Aufwand auf der Client-Seite oft auf einfache Skriptprogramme [Wal07], für die man ein Framework zur Interpretation von Repräsentationen benutzen kann oder die Repräsentation so anpasst, dass sie einfache Grammatik aufweist. Des Weiteren ist die Steuerung durch ein festes anwendungstypisches Protokoll vorgegeben, welchem der Agent folgt, indem er die clientseitigen HTTP-Anwendungen und APIs benutzt.

Auf der Server-Seite, die gut in einen Web-Server integriert werden muss, soll der Aufwand zwischen dem Web-Server als Kommunikationsendpunkt und der eigentlichen Applikationsressource minimal groß sein. Dies wird dadurch gesichert, dass auf der *CGI*-Schicht aufgesetzt wird, die üblicherweise zum Web-Server dazu gehört und noch tief genug im System ist, damit die Implementation kein „schwerer Brocken“ wird, entsprechend der Kritik im Teil 1.3.3. An dieser Stelle wird insbesondere auf Servlet-Container und generell auf Java, was die Trägheit bei der Reaktion des Systems und viele Schichten der Abstraktion mittels XML-Deskriptoren beseitigt.

Ein weiterer Grund warum REST für die Entwicklung eines Experimental-Frameworks attraktiv ist, ist die Tatsache, dass Zustandslosigkeit und Idem-

potenz erwünschte Eigenschaften (siehe 5.4.1 und 5.4.3) in einer verteilten Archiktur sind und bei diesem Architekturstil zur Unterstützung von Caching-Komponenten förderlich sind. Zustandslosigkeit gilt bei der Kommunikation per HTTP (in der puristischen REST-orientierten Seite) generell und drei der vier bekannten Protokollverben des Webs (HTTP) sind idempotent (siehe 5.7.1 und 5.8).

6.3 Bezug zur Prozesssteuerung

Man kann mit gutem Gewissen sagen, dass REST wohl durchgedacht ist und Vorteile für eine Architektur hat. Alleine am Beispiel des *World Wide Web* und des Erfolgs dieses ursprünglich aus der Forschung stammenden Systems kann man diese Vorteile als Fakt bestätigen.

REST-konforme Web-Dienste sind allerdings sehr zäh zu entwickeln und erfordern tiefe Kenntnisse in der Materie. REST ist ein Architekturstil und kein bestimmtes Protokoll, wie das oft fälschlicherweise dargestellt wird. Um Web-Dienste REST-konform zu gestalten, ist es wichtig, die Interaktion und das Verhalten der Ressourcen gleichermaßen bei diesem Architekturstil zu betrachten.

Um Prozesssteuerung und Kommunikation mittels Ressourcen zu schaffen, muss eine solche Ressource über mehr „Intelligenz“ verfügen als eine einfache Datenablage oder Speicherzelle. Dazu braucht man vor allem die Möglichkeit Datenmengen und Ordnungen auf diesen Daten schaffen. Die Ressource wird darurch in der Lage sein, zugehörige Daten in Mengen zusammenzufassen, Reihenfolgen für gesteuerte Abarbeitung ausdrücken und Information in der verteilten Architektur zu synchronisieren, um die Abarbeitung von einzelnen entkoppelten Aufgaben zu unterstützen.

Eine bekannte Ressourcenart in Form einer Datenstruktur ist eine Aufgabenliste in Form eines Newsfeed-Kanals, was man öfters in Form eines Atom-Feeds [NS05] oder eines RSS-Feeds [RSS08] umgesetzt sieht. Diese schon längst bekannte abstrakte Datenstruktur, die ihre hier Anwendung gefunden hat, war der Ansporn, sich mit Kontrolle in Prozessen auseinander zu setzen, die hier offensichtlich mit der implizit gegebenen Ordnung in der Datenstruktur modelliert wird.

Im Folgenden wird vorgestellt wie das Experimental-Framework funktioniert, in dem datenstrukturbasierte Ressourcen implementiert werden können. Im nächten Kapitel 7 wird darüber gesprochen, wie solche Art von Ressourcen aussehen kann und welche Eigenschaften sich dabei ergeben.

6.4 Schichten eines REST-Dienstes

Ein Schichtenaufbau ist für Informatiker ein sehr weit verbreitetes Mittel, um Middleware in funktionale Einheiten zu gliedern. Der Grund ist, dass die Argumentation über die Eigenschaften einer bestimmten Schicht den Entwickler davon entbindet, die anderen Schichten gleichzeitig zu betrachten. Es vereinfacht den Entwurf, die Implementierung und die Spezifikation.

Ferner werden hier Ressourcen betrachtet, denen eine Datenstruktur unterliegt und wo Zusammenhänge zwischen Daten ausgedrückt werden können. Basierend auf diesen Überlegungen würde sich das folgende Schichtenmodell empfehlen, das in Abbildung 6.1 illustriert ist.

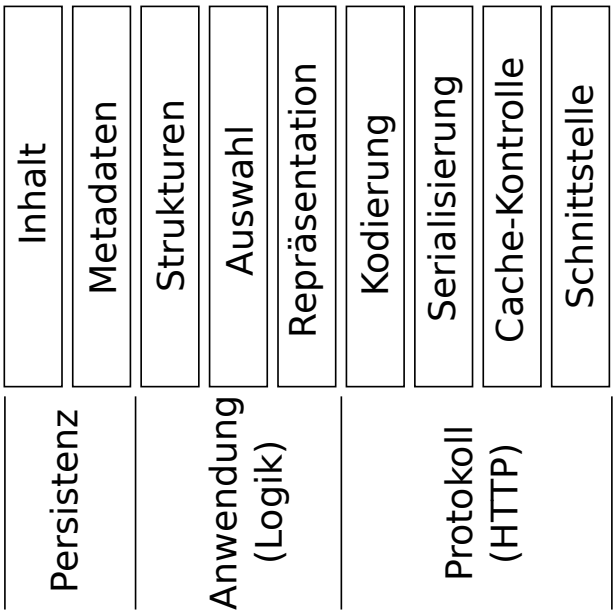

Abbildung 6.1: Interner Schichtenaufbau eines REST-Dienstes

Grob lassen sich die Schichten zu den Funktionen der Persistenz, der Anwendungslogik in der Ressource und der Protokollbehandlung (hier explizit HTTP) zuordnen. Zunächst fällt es nicht auf, dass die Protokollbehandlung sich mit der Logik des Dienstes und sogar mit der Art und Weise, wie die Persistenz organisiert konzeptuell beeinflusst. Dies ist ein Problem in der Entwicklung von Web-Diensten nach REST-Architekturstil, auf das später eingegangen wird.

6.4.1 Inhalt

Die Schicht, die am tiefsten im System ist, gehört zur Persistenz und speichert den Inhalt einer Ressource. Zum Inhalt gehören Daten, die die Ressource braucht

um die eigentlichen Daten über sich selbst an Anfragende zu liefern. Der Inhalt kann natürlich auch modifiziert werden, wenn die Ressource es zulässt. Zum Inhalt zählt der Ressourcenzustand dazu, der im Teil 5.2.2 mit Z bezeichnet worden ist.

Eine ganz besondere Form von Inhalt sind Ressourcen, die externe Systeme reflektieren. So etwas tritt zum Beispiel ein bei einer Ressource, die die aktuelle Uhrzeit angibt. Typischerweise wird eine Uhrzeit nicht in einer Datei oder einer Datenbank gepflegt und erfordert einen Aufruf im angekoppelten System (dem Zeitgeber), welcher nicht innerhalb der eigentlichen Architektur abgehandelt wird. Solche Variablen wie die hier angesprochene Uhrzeit, oder eine durch einen Sensor gemessene Temperatur, sind zwar nicht im direkten Sinne persistiert worden, aber verhalten sich wie ein persistentes Datum, das im System gespeichert wird.

6.4.2 Metadaten

Die Metadaten sind Daten, die zur Verwaltung der Ressource gehören, aber nicht in die Produktion der Repräsentation einfließen. Dazu gehört insbesondere die Cache-Verwaltung, deren Einstellungen im Protokoll mitangegeben werden, und die die eigentlichen Daten nicht braucht, um zu beurteilen, ob sich der Inhalt geändert hat. Für jeden Zustand, welcher zur Bildung einer bestimmten Repräsentation gehört, gibt es eine eindeutige Kennung an der die Caches die nichtmodifizierte Version erkennen. Im HTTP-Protokoll gibt es dazu den Kopf namens `ETag` [F+99] im Teil 14.19. Außerdem gehören zur Cache-Steuerung das Datum der letzten Änderung und die Angabe der Gültigkeitszeiträume, die für den Inhalt variabel gestaltet werden kann, und demzufolge mitgespeichert werden sollte.

Zu den Metadaten gehören keine Daten, die zur Änderung der Repräsentation führen könnten. Insbesondere müssen zwei verschiedene Agenten bei gleicher Anfrage, die gleiche Antwort erhalten, unabhängig ihrer Identität. Zu beachten ist außerdem, dass der Inhalt und die Metadaten, die einzigen persistenten Daten in der Ressource sind. Dies bedeutet, dass sie den Zustand Z dieser Ressource bilden, aus der das komplette, Verhalten der Ressource und die Repräsentation bestimmt werden muss. Alle Methoden und die darin ablaufenden Berechnungen basieren auf diesen Daten und müssen bei Bedarf in diesen beiden Schichten abgebildet werden.

6.4.3 Strukturen

Ab dieser Schicht fängt die Programmlogik des Web-Dienstes an. Die persistenten Daten, die darunterliegen werden durch diese Logik zu einer logischen Struktur zusammengefasst. Diese Schicht versteht aus welchen gespeicherten Elementen sich der Inhalt bildet und wie sie sich aus den darunterliegenden Schichten konstruieren lassen. Des Weiteren kann diese Schicht Beziehungen zwischen den Elementen nachvollziehen, sodass logische Reihenfolgen und andere Anordnungen zusammengehörig geliefert werden können.

Die Strukturschicht ist auch der Ort, wo die Implementierung bezüglich der Datenstrukturengetriebenheit, die im nächsten Kapitel 7 weiter erörtert wird. Der Ausdruck, der hier bei abstrakten Datenstrukturen oft relevant ist, ist das *logische Element*. Die Strukturschicht kann die Zuordnung zwischen den physikalisch gespeicherten Daten und dem zugehörigen logischen Element herstellen.

6.4.4 Auswahl

Die Auswahlschicht stellt unterstützende Dienstmethoden zur Verfügung, um bestimmte Teile des Ressourceninhalts, der eine Struktur durch die darunterliegende Schicht bekommen hat, zu ermitteln. Sie ist in der Lage die logischen Elemente auszuwählen, die für die Erstellung der Anfrageantwort von Bedeutung sind. Konkret bietet die Auswahlschicht Methoden zum Auffinden von Elementen und der Suche nach Elementenmengen. Sie kann einzelne Elemente mit anderen kombinieren, um so etwas wie Pagination zu erzeugen, also das „Blättern“ innerhalb von Suchergebnissen.

Nebenbei angemerkt, es ist möglich, die Auswahl und die Struktur in eine einzelne Schicht zusammen zu fassen. Die Mechanismen dieser Schicht sind aber von der Programmlogik gesehen höherwertig, als die der Strukturschicht. Die Trennung der Auswahlschicht von der Stukturschicht kann die Wiederverwendbarkeit der Auswahlschicht fördern.

6.4.5 Repräsentation

Die Schicht der Repräsentation hat die Aufgabe, die anfragten logischen Strukturen in eine für den anfragenden Client verständliche, zuvor vereinbarte Form, zu bringen. Aus der logischen Darstellung im Speicher wird als eine konkrete Repräsentation in einem direkt abspeicherbaren Format.

Diese Schicht kann zumeist nicht anwendungsneutral erstellt werden. Das bedeutet, dass oft ein vorgefertigtes Muster verwendet wird, in Form einer Vorla-

ge, und eine Zuordnung von Datenstrukturen zu gleichfömig zu produzierenden kompletten Repräsentation geschieht. Eine Möglichkeit ist, hier Template-Prozessoren (auch genannt „Template-Engines“) einzusetzen oder die Produktion der Repräsentationsausgabe einfach auszuprogrammieren.

Es ist üblich, die erstellte Repräsentation im Speicher zu puffern, weil anhand dieser protokollspezifische Angaben ermittelt werden (zum Beispiel die Größe der Übertragung in dem `Content-Length`-Kopf [F+99]), die erst nach der Erstellung der kompletten Repräsentation feststehen.

6.4.6 Kodierung

Die Kodierungsschicht ist gegenüber der Repräsentation agnostisch und bereitet die Repräsentation für die Übertragung vor. Dazu gehören Mechanismen wie Kompression und Zerlegung in Übertragungsblöcke („chunked encoding“). Da die Kodierungsschicht hinreichend abstrakt ist und nicht anwendungsspezifisch, wie die Repräsentation, sollte man diese Schichten nicht miteinander vermischen.

Auch die nachfolgende Schicht sollte man getrennt lassen, denn nach der Kodierung erst steht die Protokollantwort in der Form fest, wie Caches sie speichern würden. Auf dieser Basis lässt sich nach der Kodierung das `ETag` erst vollständig berechnen.

6.4.7 Serialisierung

Hat man die Repräsentation in kodierter Form muss diese stückchenweise, durch das Protokoll übertragen werden. Hier kommt die Serialisierungsschicht zum Einsatz, die die zu sendende Repräsentation in einen Datenstrom umwandelt und diesen Portionsweise überträgt. Diese Schicht behandelt auch eventuelle Verbindungsfehler und bricht die Übertragung geeignet ab.

Diese Schicht erlaubt ebenfalls die Wiederaufnahme eines Datenstroms ab einer bestimmten Position, die zuvor aus technischen Gründen unterbrochen worden ist. Auf diese Weise verkürzt sich die Übertragungszeit und die Netzwerkauslastung.

Wie man leicht sehen kann fällt die Serialisierung bereits unter den Aspekt der Kommunikation und ist gleichzeitig die sich am tiefsten befindliche Kommunikationsschicht.

6.4.8 Cache-Kontrolle

Die Cache-Kontrollschicht wertet konditionale Anweisungen aus, die bei der Anfrage im Kopf mitübergeben werden. Auf dieser Schicht ist es möglich vorab zu bestimmen, ob eine Anfrage vollständig abgearbeitet werden soll, oder ob sie an die sich höher befindliche Cache-Hierarchie zurück gereicht wird.

Das Caching profitiert von der Tatsache, dass die Antwort auf eine fest vorgegebene Leseanforderung beim Web-Dienst unabhängig vom Client stets gleich ist. So können vorgeschaltete Caches die Antworten auf Anfragen ohne die Belastung des eigentlichen Web-Dienstes abhandeln.

Die Cache-Kontrollschicht muss, um Rechenaufwand verringern zu können, feststellen können, ob der ursprüngliche Zustand der Ressource sich gegenüber der konditional gestellten Anfrage, die sich an früheren Zuständen orientiert, verändert hat. Das bedeutet, dass die Cache-Kontrollschicht direkt diese Änderung feststellen muss, ohne Beteiligung der Kontrollschichten.

Es empfiehlt sich also, die eindeutige Kennung `ETag` so zu gestalten, dass ein Teil der Kennung sich ausschließlich auf den Zustand Z der Ressource bezieht und im Rest der Kennung erst nach der Kodierungsschicht entsteht und klar getrennt in der Kennung auftaucht. Dies ist ein Hinweis, der weder bei der Spezifikation [F+99], noch bei der Beschreibung von REST gegeben wird und basiert auf den Überlegungen, die zu dem *cgi2c*-Framework angestellt worden sind.

Hinzuzufügen ist noch, dass Caching vorwiegend bei lesenden Anfragen zur Effizienz beiträgt. Also vor allem Anfragen, die „safe“ sind. Dazu gehören ebenfalls (unter gewissen Umständen) Fehlermeldungen, wie man im Teil 5.2.2 feststellen konnte. Die Bedingung dabei ist, dass der Fehler eine gewisse Zeit in der Zukunft eintreten wird. HTTP unterstützt dazu eine Einteilung in temporäre und permanente Fehler.

Wie lange ein Zustand gültig ist oder ob der Zustand sich bereits verändert hat, wird in der Metadatenschicht gespeichert. In der Cache-Kontrollschicht, die die Einschätzungen bezüglich Caching intepretiert, wird diese Information aus den Metadaten in protokollspezifische Angaben (hier HTTP und Caching-Köpfe) übersetzt.

6.4.9 Schnittstelle

Die oberste Schicht des Web-Dienstes ist seine Schnittstelle, die den Web-Dienst, und die darin gekapselte Ressource, mit dem Server und anderen zwischenliegenden diensteneutralen APIs (dem Framework) verbinden.

In der Schnittstelle wird festgelegt, welche Methoden angeboten werden, welche Parameter sie haben und wie sie ins Framework eingebunden werden. Das überliegende Framework stellt Mechanismen zur Verfügung, die eingehende Anfrage bezüglich des HTTP-Protokolls zu untersuchen, vorzubereiten und den richtigen Web-Dienst, mit der richtigen Methode und der verlangten Repräsentation aufzurufen, welcher angefragt worden ist („routing“). Diese Schnittstellenschicht übernimmt dann den dienstespezifischen Teil der Arbeit. Das bedeutet, sie interpretiert den parametrisierten Aufruf und stellt die Mittel zur Kommunikation zur Verfügung (Puffer und andere Einstellungen). Es werden Anfrageköpfe interpretiert, die unterliegende Schicht des Dienstes aufgerufen und Antwortköpfe erstellt.

6.5 Abhängigkeiten zwischen den Schichten

Die Abbildung 6.2 zeigt Abhängigkeiten zwischen den Schichten, im Falle einer Änderung des Web-Dienstes bei der Entwicklung. Sie ist wie folgt zu lesen. Ändert man in der Implementierung die Schicht in einer Zeile dieser Matrix, *muss* (oder *kann*) das dazu führen, dass die Implementierung der Schicht in einer Spalte angepasst werden muss. Das *Muss* wird in der Matrix durch einen ausgefüllten und das *Kann* durch einen leeren Kreis ausgedrückt. Ein leeres Feld bedeutet, dass bei ordnungsgemäßer Implementierung keine Anpassung nötig ist.

Erfreulich ist, dass unterhalb der Diagonale keine Abhängigkeiten auftauchen. Dies wäre ein enormes Problem, denn es würde bedeuten, dass es Rückwärtsabhängigkeiten bei den Schichten gibt und eine Anpassung der höherliegenden Schicht, den Entwickler wieder zur Anpassung von tieferen Schichten zwingen würde.

Auf der linken Seite der Spalte *Kodierung* stellt man fest, dass sehr starke Beziehungen zwischen dem Inhalt und den anwendungsorientierten Schichten bestehen. Die *Schnittstelle* als Eintrittspunkt des Web-Dienstes reflektiert diese Abhängigkeiten ebenfalls. Alle diese Schichten zusammen gehören zu dem was die Anwendung eigentlich an Funktionalität anbieten sollte. Da eine Ressource stark datenbasiert ist, haben die gespeicherten Daten selbst (unter *Inhalt*) den stärksten Einfluss auf den weiteren Entwurf des Dienstes.

Die Schichten der *Kodierung* und *Serialisierung* sind relativ neutral gestaltet und können meistens eigenständig funktionieren. Das liegt daran, dass die Art wie Schichten gestaltet sind, die Benutzung von anderen Hilfsbibliotheken erlaubt (Komprimierverfahren und Pufferung). Es kann aber vorkommen, dass bestimmte Kodierungsarten andere Serialisierungsformen bedürfen, was mit einer schwachen Abhängigkeit annotiert wurde.

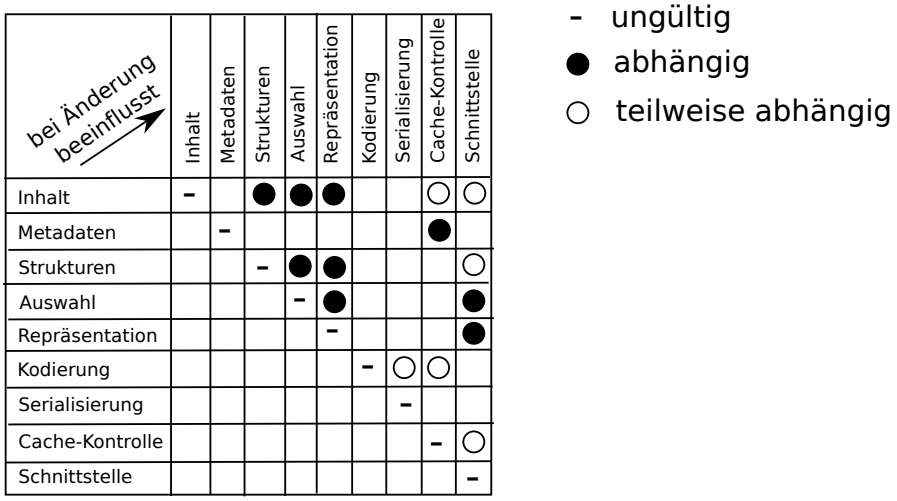

Abbildung 6.2: Abhängigkeiten zwischen Implementationsschichten eines Web-Dienstes

Eine der komplexesten Sachverhalte hat der Entwickler eines Dienstes im Zusammenhang mit Caching. Es wurde schon zuvor gesagt, dass der Cache mit den Persistenzschichten, allem voran der Metadatenschicht, eng verbunden ist. Würde die Cache-Kontrollschicht die Anwendungsschichten gebrauchen, bedeutete das, dass der Aufwand, konditionale Anfragen zu machen dem Aufwand der einfachen Anfrage entspricht und es würden sich keine Vorteile bezüglich Last ergeben, wenn der interne Ressourcenzustand sich nicht ändert. Wie auch schon vorher erwähnt wurde, beeinflusst die Kodierungsschicht die Cache-Kontrolle, weil nach der Kodierung erst das `ETag` komplett bekannt ist. Dieser Zusammenhang kann die Implementierung der Cache-Kontrolle beeinflussen.

Die Cache-Kontrolle kann in vielen Fällen eine Auswirkung auf die Beschreibung der Schnittstelle haben. Schon dort muss unter Umständen entschieden werden in welchen Fällen Caching auf welche Art und Weise gemacht werden soll und wann nicht.

6.6 Resultierende Implementierung

Nach den zahlreichen konzeptuellen Überlegungen wurde schon im Juni 2009 das erste Mal mit einer Implementierung des Experimental-Frameworks angefangen. Dieses basiert auf REST-Prinzipien umgesetzt auf HTTP und wird somit

an den Web-Server mit Hilfe der CGI-Schnittstelle direkt angebunden. Diesen Sachverhalt illustriert die Abbildung 6.3.

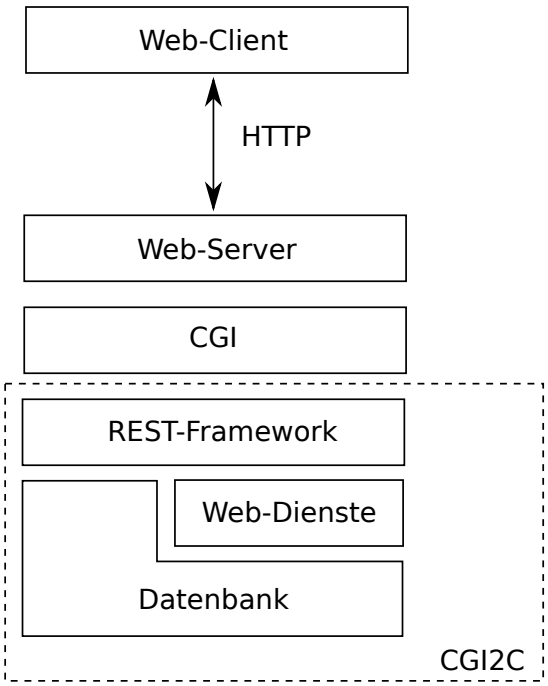

Abbildung 6.3: CGI2C Komponentenarchitektur

Wie schon am Anfang des Kapitels angemerkt, wird bei der Implementierung auf Zwischenkomponenten verzichtet, wie zum Beispiel den Servlet-Container und andere Formen von Middleware. Das Framework „cgi2c“ ist komplett in der Programmiersprache C umgesetzt worden und nutzt für die Implementierung der Web-Dienste den C-Präprozessor [CPP05] und die Fähigkeit, den eigentlichen Dienst erst per Late-Binding mit Hilfe von Shared-Libraries zu laden und dann auszuführen. Nach diesem Verfahren kann das REST-Framework von den Web-Diensten selbst separiert werden. Dies führt dazu, dass Web-Dienste, die später die Ressourcen realisieren sollen, eigenständig entwickelt werden können.

Die Entwicklung von Web-Diensten wird auch dadurch unterstützt, dass eine rudimentäre Datenbank-Schicht zur Persistierung der Zustände direkt zur Verfügung gestellt wird, sodass die Dienste keine weiteren Maßnahmen für die Zurverfügungstellung der Persistenz brauchen. Da die Web-Dienste Anwendungsbibliotheken sind, ist es natürlich möglich, diese beliebig mit anderen Persistierungsmechanismen zu erweitern.

Nachfolgend werden zwei Dienste näher erklärt, die andere Web-Dienste im Framework *cgi2c* verwalten und deswegen eine Sonderstellung einnehmen.

6.6.1 Der Web-Dienst „Binder“

Ein ganz besonderer Dienst ist der *Binder*. Der Binder ist für die Zuordnung von URIs zu Web-Diensten zuständig, die vom Framework nachgeladen werden, wenn der Web-Dienst eine Anfrage bekommt. Um das Konzept in sich geschlossen zu halten kann der Binder wie ein gewöhnlicher Web-Dienst benutzt werden (im Web) werden. So ist man in der Lage mit einem gewöhnlichen Browser, die verfügbaren Dienste an URIs anzubinden, indem man Formulare ausfüllt. Der Binder ist in der Abbildung 6.4 dargestellt. Insbesondere kann man den Binder selbst an URIs binden, was die Verwaltung flexibilisiert. Die einzige Voraussetzung ist, dass mindestens ein funktionierender Binder im Framework erhalten bleibt. Dies wird dadurch sichergestellt, dass der Binder, der gerade aktiv ist, sich nicht selbst aus dem Framework entfernen kann. Das Framework würde auf jeden Fall funktionieren, aber für Experimentalzwecke wäre es ungünstig, wenn man keine Einstellungen beim Framework mehr machen kann.

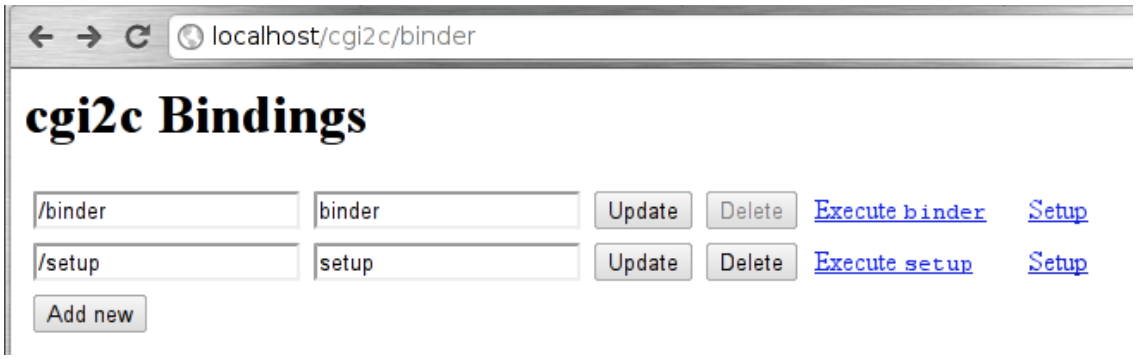

Abbildung 6.4: Der Web-Dienst Binder *im Web-Browser.*

Die Abbildung 6.4 zeigt die gebundenen Anwendungen im Framework *cgi2c*. Im linken Textfeld ist der Pfad und im rechten der Name des an diesen Pfad gebundenen Web-Dienstes. In diesem Formular ist der Pfad, an dem die Anwendung angebunden ist modifizierbar. Man kann die Web-Dienste auch hier löschen, wobei der laufende Binder (vergleiche *URI* oben mit Pfad im Formular) nicht entfernt werden kann.

Mit der Verknüpfung „Execute ...“ kann die Anwendung mittels des HTTP-Verbs *GET* und ohne Angabe von Parametern ausgeführt werden. Viele Anwendungen würden hinter diese Verknüpfung eine Darstellung wählen, die ein Mensch am Browser verstehen und bedienen kann. Es ist jedoch davon auszugehen, dass auch andere HTTP-Verben im Web-Dienst existieren oder bestimmte Parameter übergeben werden müssen, sodass ein Teil der Funktionalität des jeweiligen Web-Dienstes auf anderen Wegen zugänglich ist.

Die zweite Verknüpfung „Setup“ (rechts in der letzten Spalte) erlaubt die Konfiguration des an den Pfad gebundenen Web-Dienstes mittels des Web-Dienstes *Setup*. Der Knopf „Add new“ (unten im Bild) erlaubt das Anlegen eines neuen Web-Dienstes unter einer angegebenen URI, siehe Formular in Abbildung 6.5.

Abbildung 6.5: Das Anlegen eines neuen Web-Dienstes im Binder.

6.6.2 Der Web-Dienst „Setup“

Dieser Dienst ist zwar optional, wie jeder andere Dienst im Framework, aber er erlaubt Parameter für andere Web-Dienste zu modifizieren, was ihn zu einem der wichtigeren Dienste macht, die den Binder unterstützen. Wichtig ist zu bemerken, dass *Setup* nicht benötigt wird, um die Funktionalität der Parameterübergabe für die Web-Dienste auszuführen, die man schon beim Aufruf des konkreten Dienstes eingestellt hat, weil dies die Aufgabe des Frameworks ist. Somit ist *Setup* in der gleichen Rolle wie *Binder*, nur für die Änderungen der Einstellungen zuständig.

Setup übernimmt hier die Rolle der Dienst-Deskriptoren [XML02] [Sha09], die bei Java-Enterprise-Applikationen nötig sind, um initiale Einstellungen festzulegen. Die Parameter des Web-Dienstes sind nicht das gleiche wie die Benutzerparameter, die der Dienst beim Aufruf einer Methode bekommt. Die ersteren reflektieren serverseitige Voreinstellungen. Die letzteren gehören protokollseitig zur Anfrage, die der Benutzer des konkreten Dienstes stellt.

In Abbildung 6.6 wird beispielhaft dargestellt wie der Web-Browser den *Setup*-Web-Dienst ausführt, um den Parameter `destination` für den Web-Dienst *redirect* (gebunden an URI `/redirect`), festzulegen. Die Web-Anwendung *redirect* leitet den Benutzer eines Web-Browsers von ihrem Aufrufpfad auf das im Parameter `destination` spezifiziertes Ziel (hier wurde `http://www.google.de/` angegeben).

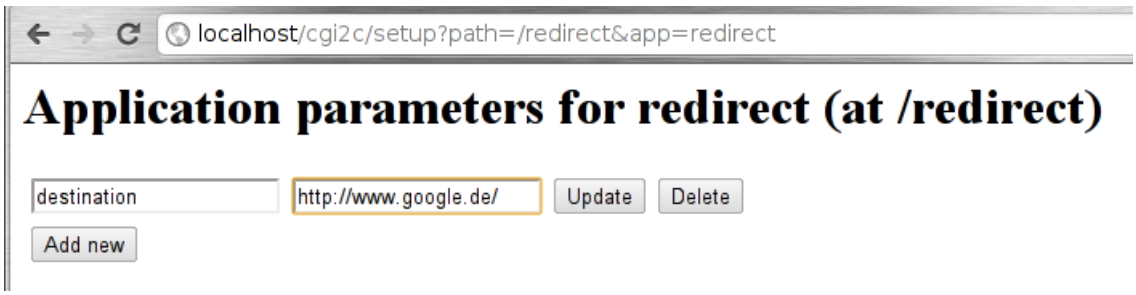

Abbildung 6.6: Das Einstellen von Parametern eines Web-Dienstes in Setup.

6.7 Dienstearchitektur im Framework *cgi2c*

Im Teil 6.6.2 wurde bereits ein Web-Dienst namens *redirect* erwähnt. Dies ist der einzige bis jetzt vorgestellte Dienst, der mit dem Management des Frameworks nicht zu tun hat und damit als eine eigenständige Dienstanwendung auf der Basis des Frameworks *cgi2c* entwickelt wurde. In diesem Teil wird die Entwicklung von solchen Web-Diensten näher beleuchtet.

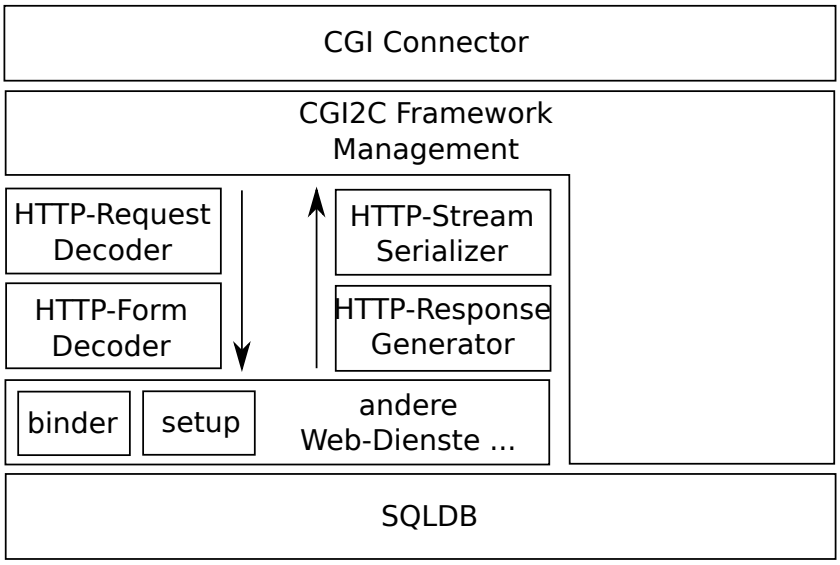

Abbildung 6.7: CGI2C Dienstearchitektur

In Abbildung 6.7 sieht man die Beziehung zwischen den programmierbaren Web-Diensten und den umliegenden Komponenten des Frameworks. Eine Anfrage wird über den *CGI-Connector* empfangen und durch das *CGI2C Framework Management* analysiert. Diese Management-Komponente ermittelt anhand von den in der Datenbank (*SQLDB*) enthaltenen Binding-Informationen den zu benutzenden Web-Dienst. Der Web-Dienst wird anschließend geladen, initialisiert und das „Routing“ der Methoden festgelegt. Die Anfrage wird dann an den

HTTP-Request Decoder weiter gereicht, welcher die Anfrage interpretiert und in Form von Applikationsstrukturen an den Web-Dienst weiter gibt. Der HTTP-Request Decoder kann in einer Unterkomponente namens *HTTP-Form Decoder* feststellen, ob zusätzliche Informationen in Form von Formulardaten im Körper der Anfrage übergeben worden sind und diese ebenfalls für den Zugriff durch den Web-Dienst vorbereiten.

Alle zuvor ermittelten Informationen können so an die Anwendung überreicht werden, indem sie als Parameter beim Aufruf der Dienste-Methode angegeben werden. Die Anwendung wird vom Management anschließend gestartet und führt ihre Arbeit durch. Während der Ausführung wird in vielen Fällen eine Antwort auf die Anfrage konstruiert. Der *HTTP-Response Generator* stellt Routinen zur Verfügung, um diese Antwort bequem erstellen zu können. In den meisten Fällen ist eine Antwort anhand von den Daten aus der Persistenz, hier *SQLDB*, zu erstellen. Man kann im Web-Dienst selbst dabei vollständig auf die Initialisierung der Datenbankverbindung verzichten, denn die Sitzungsverbindung wird von der Management-Schicht an den jeweiligen Web-Dienst zur sofortigen Verwendung weiter gereicht. Wurde die Antwort, die die Methode erzeugt hat, fertiggestellt und optionale Steuerungsinformationen gemacht, ist die Antwort durch die Komponente *HTTP-Stream Serializer* zu serialisieren und dann zurück an den HTTP-Server zu übergeben. Die Management-Schicht übernimmt dabei die Aufräumarbeit und beendet den vom Web-Server ausgeführten Prozess.

6.8 Status der Implementierung

Das Framework hat sich als sehr nützlich erwiesen, um schnell einige Experimente mit datenbasierten Ressourcen durchzuführen. Die Ressourcen können ohne größeren Aufwand erstellt und an mehreren *URI*s unabhängig betrieben werden.

Die Implementierung leidet zur Zeit noch aus der Perspektive der Performanz an Latenzproblemen, weil jeder Aufruf das Framework erst initialisieren muss, bevor die Applikation starten kann. Das Framework wird auch nach der Anfrage beendet und muss neu geladen werden. Hier gibt es Potenzial zur Optimierung, indem die Applikation im Betrieb die Anfragen annimmt und die Initialisierungsphase nur ein Mal durchführt. Dies erfordert jedoch einen Web-Server-spezifischen Anbindungsmechanismus im Gegensatz zu dem gewählten standardisierten *CGI*.

Im Teil 6.4.8 wurde ausführlich dargelegt wie man Caching in das Framework einbauen sollte, damit Last von Ressoucen genommen werden kann. Zu diesem

Zeitpunkt wurde die Caching-Schicht noch nicht realisiert und zwingt den Entwickler, eine selbstgebaute Lösung zu schreiben. Es ist durchaus möglich, das Framework diesbezüglich zu erweitern, aber dies erwies sich nicht als nötig, um es für die experimentelle Auswertung der in späteren Kapiteln untersuchten Protokoll- und Ressourceneigenschaften zu benutzen.

Sicherlich ist die prototypische Implementierung von *cgi2c* noch nicht zufriedenstellend. Es fehlt an Funktionalität und einige Konzepte, die recht unsauber sind, müssen überarbeitet werden. Insbesondere empfiehlt es sich bezüglich Sicherheit die Zeichenkettenverarbeitung robuster zu gestalten, weil dies generell ein Sicherheitsrisiko ist. Denkbar wäre an dieser Stelle eine striktere Typenbehandlung, um festzustellen ob Zeichenketten für die Datenbank, für URIs, HTTP-Köpfe und die Repräsentation sicher nutzbar sind. Zur Zeit muss das nach gutem Gewissen manuell durch den Entwickler gesichert werden, was potenziell eine Fehlerquelle ist, aus der Sicherheitslücken entstehen können. *cgi2c* wurde nicht mit dem Ziel entwickelt, es sicher zu gestalten, weil der Schwerpunkt auf den Experimenten mit Ressourcen und Protokollen liegt.

6.9 Vergleich der Performanz mit Jersey

cgi2c und die bekannte auf Java basierende Referenzimplementation *Jersey* lassen sich nur schwierig vergleichen, weil sie vom Grund auf anders funktionieren. Während *cgi2c* auf der freien Software *Apache HTTP-Server* [Apa12a] läuft und dort bei Bedarf erst geladen wird, ist *Jersey* und die Dienste direkt in die Anwendung statisch in den Web-Container eingebunden. Dies führt dazu, dass das Framework, dass es sich in Sachen Performanz wenig von der herkömmlichen Java-basierten Lösung unterscheidet.

Der Gleichstand bei der Performanz wird aber bei der Jersey-Lösung damit bezahlt, dass Dienste sich nicht dynamisch von der Oberfläche aus konfigurieren lassen. Sie sind an feste Pfade gebunden und bleiben dort auch, bis der Web-Container neu startet oder, falls bessere Web-Container, wie Tomcat [Apa12b] verwendet werden, der Web-Dienst mit anderen Einstellungen in XML-Deskriptoren unter anderem Pfad neu installiert wird. Insgesamt sieht man, dass es einfacher ist, Ressourcen bei *cgi2c* an verschiedenen URIs schnell anzuhängen. Dabei muss man den Quelltext für den Web-Dienst nicht mehr ändern oder manuell auf dem Datenträger kopieren.

Einige ausgiebige Tests haben gezeigt, dass *cgi2c* besser parallelisierbar ist als eine Vergleichsimplementation auf Jersey und dem Servlet-Container *Grizzly* [Gri11], die in Java programmiert wurde. Die Java-Implementierung hatte Probleme mit dem Kontakt zur Datenbank als die Zahl der Verbindungen im Pool

übergelaufen ist. Das Problem ist sowohl während stark parallelisierter als auch bei der Hintereinanderausführung von Anfragen aufgetreten. Der Web-Dienst hatte reproduzierbar mit Überlastung zu kämpfen und fiel zeitweise aus, bis sich der Verbindungs-Pool wieder gefüllt hat. Es hat auch dazu geführt, dass viele *GET*-Anfragen auf *Jersey* nicht mehr ordnungsgemäß beantwortet wurden, als Zeitweise keine Verbindungen zur Datenbank aufgebaut werden konnten. Bei Experimenten sollten Frameworks ein vorhersagbares Verhalten aufweisen und solche Situationen möglichst nicht auftreten.

Ein weiterer Vorteil, der eher für den Realeinsatz relevant ist, ist dass durch die Verzicht auf schwergewichtige Komponenten der Speicherverbrauch deutlich geringer ist. Insgesamt nimmt das Framework während der Laufzeit etwa ein Zwanzigstel des Speicherplatzes im Vergleich zur Java-Anwendung, die nach und nach auf 222 Megabyte Speicherverbrauch angewachsen ist.

6.10 Vergleich des Implementierungsaufwands

Es wurde schon erwähnt, dass die Konfiguration bei *cgi2c* bequemer, vom Web aus mit wenig Aufwand per Web-Formular gemacht werden kann. Wie sieht es jedoch mit der Programmierung eines Web-Dienstes?

Die Länge der Implementierung ist effektiv gleich. Die Programmiersprache C ist grundsätzlich schwieriger zu benutzen als Java. Deswegen wurde bei *cgi2c* darauf geachtet, dass möglichst viele nützliche Funktionen dem Entwickler zur Verfügung stehen. Das fängt damit an, dass die Bibliothek *glib* benutzt wird, um abstrakte Datenstrukturen nutzen zu können, für bessere Zeichenkettenverarbeitung und einfacheres Speichermanagement. Zudem wird dem Nutzer erspart, sich um die Initialisierung von Datenbankverbindungen zu kümmern. Es kann sofort mit der Datenbank interagiert werden. Da die Programmiersprache C keine Annotations mitbringt, jedenfalls nicht ohne externe Werkzeuge, auf die hier absichtlich verzichtet worden ist, wurden für die Deklarationen der Methoden und ihre Registrierung im Framework stattdessen Makros benutzt, die durch den gewöhnlichen C-Präprozessor behandelt werden. Auf diese Weise wird eine kleine Portion an Quellcode eingespart, der für den Entwickler, später beim Kompilieren, generiert wird.

Ein Implementationsziel bei *cgi2c* war die einfache Wiederverwendbarkeit der Dienste, um ein gleiches Verhalten für mehrere Ressourcen spezifizieren zu können. Somit ist es möglich, Experimente mit wenig Aufwand beim Aufbau zu unterstützen. Die als Alternativen vorgestellten REST-Implementationen binden hingegen die Dienste an festgelegte *URI*s mit Hilfe von Annotationen im Quell-

text, was die Wiederverwendbarkeit behindert. Dies würde stets auf umständliche Vervielfachung von Quellcode hinauslaufen.

Auch das Installieren eines Web-Dienstes gestaltet sich einfacher bei *cgi2c*. Alle nötigen Informationen zur Funkionalität liefert eine einzige Datei und das ist die Bibliothek, die an einen festgelegten Installationsort auf dem Datenträger kopiert werden muss. Ab diesem Zeitpunkt ist der Web-Dienst im *Binder* verfügbar und kann dort an eine URI gebunden werden. Hingegen muss der Web-Dienst unter *Jersey* entweder, im Falle des Gebrauchs des Web-Containers *Grizzly* bereits vor dem Start des Servers kompiliert am richtigen Ort liegen, oder im Fall des Servlet-Containers *Tomcat* in Form eines *WAR-Archivs* in ein Installationsverzeichnis kopiert werden. Möchte man den gleichen Web-Dienst an verschiedenen Pfaden binden, bedeutet das, dass jede Installation des Web-Dienstes einzeln angepasst werden muss. Bei *cgi2c* muss lediglich die eine Bibliothek ersetzt werden, in der das Verhalten korrigiert wurde. Dies wirkt sich auf alle URIs gleichzeitig aus, an denen die Anwendung gebunden ist.

Für den späteren Einsatz und Entwicklung von Agenten hat es sich als nützlich erwiesen, die Informationen in Kopf der HTTP-Antwort möglichst vollständig zu liefern. Schaut man sich die generierten Antworten der beiden Frameworks an, stellt man fest, dass *cgi2c* hier sehr viel genauer spezifiziert, wie die Antwort zu verstehen ist. Insbesondere wird die Länge der HTTP-Antwort im Standardfall berechnet und an den Client geschickt, was nützlich ist, wenn sich dieser auf den Empfang der Antwort zunächst vorbereiten muss (Pufferreservierung). Diese Genauigkeit bezahlt man natürlich mit Performanznachteilen, die hier aber gerne in Kauf genommen werden, um reibungslose Interaktion zwischen Ressourcen und Agenten zu gewährleisten.

6.11 Beispiel für einen Web-Dienst

Abschließend, zum besseren Verständnis der Funktionalität, wird hier ein kurzes Beispiel aufgeführt, um die Verwendung des Frameworks zu demonstrieren.

In *cgi2c* ist ein Dienst relativ einfach zu schreiben und zu installieren. Das hier aufgeführte Beispiel, im Listing 6.1 besteht aus 2 Teilen. Aus der Definition des Web-Service-Verhaltens einer Routine (Zeilen: 10 bis 17) und aus der Zuordnung dieser Routine zu einem HTTP-Verb und einem Datenformat anhand der MIME-Bezeichnung (Zeilen: 22 bis 26).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #include <application.h>
```

```
6
7  /*
8          Verhaltensspezifikation des Web-Dienstes auf das HTTP-Verb GET.
9  */
10 CGI2C_DECL_METHOD(helloworld_get,env,resp,db) {
11
12         stream_open(resp, HTTP_STATUS_OK, 1);
13         stream_printf(resp, "<html><head><title>Hello world!</title></head>"
14                 "<body><h1>Hello world!</h1></body>\n");
15         stream_close(resp);
16         return HTTP_STATUS_OK;
17 }
18
19 /*
20         Zuordnungen fuer den Web-Dienst.
21 */
22 CGI2C_DECL_MAPPINGS {
23
24         CGI2C_REGISTER_METHOD(helloworld_get, HTTP_METHOD_GET, "text/html");
25         return 0;
26 }
```

Listing 6.1: helloworld.c

Die Zeile 10 definiert die Methode `helloworld_get` mit Hilfe des Makros `CGI2C_DECL_METHOD`. In den Zeilen 12 bis 15 wird der Ausgabepuffer `resp` initialisiert und mit dem HTML-Dokument-Text gefüllt. Die gepufferte Ausgabe ermöglicht die korrekte Bestimmung der Antwortgröße, die für die Rückgabe empfehlenswert ist. Die Zuordnung der Methode `helloworld_get` zum HTTP-Verb *GET* und dem Antwortformat im MIME-Typ „`text/html`“ geschieht in Zeile 22 im Makro `CGI2C_DECL_MAPPINGS`.

Dieser Web-Dienst muss als dynamische Bibliothek kompiliert werden und die resultierende dynamische Bibliothek in das Web-Dienste-Verzeichnis des Frameworks kopiert werden. So kann er im *Binder* direkt an eine URI gebunden werden (siehe Abbildung 6.8) uns ist sofort unter der zuvor angegebenen Adresse einsatzbereit, wie die Abbildung: 6.9 zeigt.

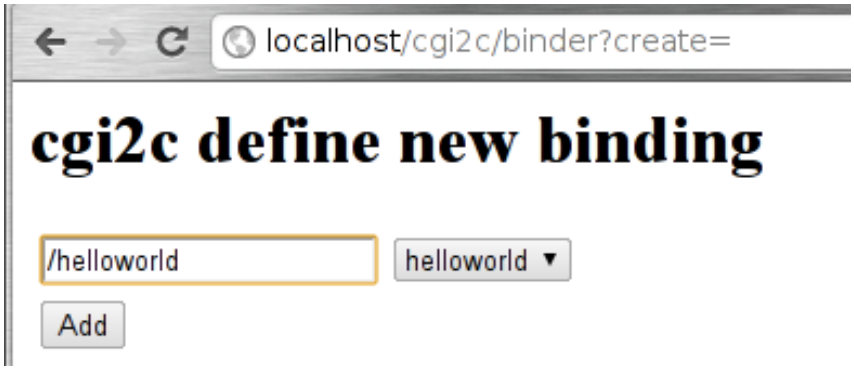

Abbildung 6.8: Festlegen der URI für den neuen Web-Dienst.

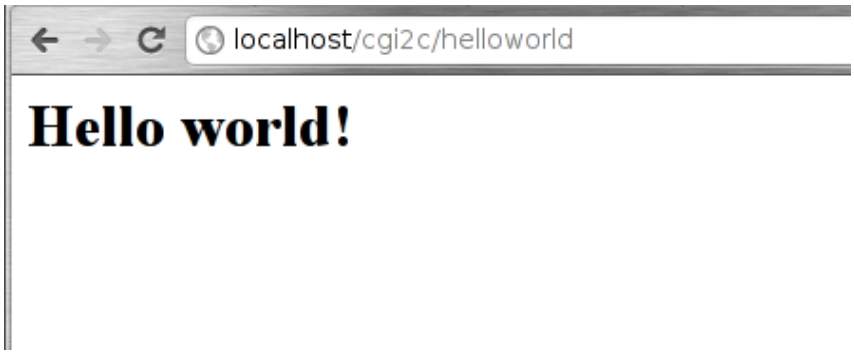

Abbildung 6.9: Aufruf des neuen Web-Dienstes im Web-Browser.

Kapitel 7

Datenstrukturen für Ressourcen

Die Idee, abstrakte Datenstrukturen zur Kontrolle in einem Programm zu nutzen, ist zwar allseits bekannt, wurde aber nicht näher für verteilte Prozesskontrolle untersucht. Das am weitesten verbreitete Beispiel für die Programmsteuerung ist der Keller für rekursive Aufrufe von Unterprogrammen. Die Möglichkeit, Abläufe durch Warteschlangen zu steuern, liefert ein einfaches Mittel um Ausführungsreihenfolgen in Form einer Ordnungsrelation auszudrücken. In diesem Kapitel wird gezeigt, dass sich diese Art von Ordnungen auf verteilte Prozesssteuerung übertragen lassen, indem man Idempotenz und Monotonie als Mittel zur Organisation von Redundanz und Fortschritt nimmt.

Wie man es aus vorhergehenden Kapiteln sehen kann, sind Ressourcen in einem ressourcenbasierten System grundsätzlich nicht in der Komplexität beschränkt. Es ist aber erwünscht, dass eine datenbasierte Komponente, zum Beispiel in einem Web-Server, bei der Berechnung der Repräsentation und bei der Ermittlung der dazugehörigen Eingabedaten, nicht zu sehr mit Rechenlast belegt wird. Deswegen wird in diesem Teil versucht, die Verwaltung minimal in Bezug auf die Laufzeit zu gestalten. Andererseits kann man als Persistenzschicht für die Ressource aber auch große Speichersysteme anbinden, sodass der Speicherplatzverbrauch unwesentlich ist.

Die Einschränkung der Rechenkraft ist aus zweierlei Gründen in den Architekturen gegeben, mit denen sich die Arbeit hier beschäftigt. Erstens handelt es sich um ein Request-Reply-Mechanismus, dessen Methoden möglichst schnell ein Ergebnis liefern sollen. Zweitens sind Arbeiten, die bei denen Rechenlast erforderlich ist, auf Agenten zu verlagern oder von der Ressource, die die Methoden abhandelt in tieferliegende Systeme abzukoppeln. Wie man das zu verstehen hat, wird im Teil 7.12 genauer erklärt.

In diesem Kapitel findet sich ein Ausblick auf Ressourcen unter dem Aspekt der Datenorganisation für Prozesszustände. Wählt man die richtigen Datenstruktu-

ren für das Management von Prozesszuständen, kann man in Systemen mit frei gestalteter Entkopplung bestimmte resultierende Eigenschaften beobachten, mit denen man Abläufe steuern kann, die sich aus entkoppelten Aktionen zusammensetzen. Die Problematik wurde im Teil über Entkopplung schon eingeführt (siehe: 3.9). Die Art und Weise wie Datenstrukturen dazu eingesetzt werden können, wird in diesem Kapitel vorgestellt. Der effektive Nutzen dieser speziell entwickelten datenstrukturgetriebenen Ressourcen wird in den nachfolgenden Kapiteln 8 und 9 näher an konkreten Beispielen erläutert.

Zunächst behandelt dieses Kapitel einige einfache Ressourcendatenstrukturen, dann wird erörtert, was man mit lastintensiven Ressourcen machen kann. Um Datenstrukturen zur Kontrolle in ressourcenbasierten Architekturen zu motivieren, muss man sich zunächst damit beschäftigen, was mit einer sogenannten *generischen Ressource* möglich ist, die keine weiteren Steuerungsmittel benutzt außer der Speicherung selbst.

In diesem Kapitel wird das Verhalten des Systems aus der Sicht der Ressourcen betrachtet. An vielen Stellen werden Beispiele gegeben, die Protokolle durch Restriktionen spezifizieren. Das bedeutet, dass eine *Auswahl* der zur Verfügung stehenden Methoden angegeben wird. Die Ressource hat durch diese Auswahl die Möglichkeit, den Agenten in der Aufrufreihenfolge zu beschränken. Der Agent hat die Möglichkeit, die verfügbaren Methoden auszuwählen. Das Agentenprogramm für datenstrukturgetriebene Ressourcen und die Regeln für das Verhalten von Agenten folgen im nächsten Kapitel. Dort wird erst aus der Sicht der Agenten und über Protokolle argumentiert.

7.1 Ressource ohne Datenstruktur: generische Ressource

Die *generische Ressource* verhält sich wie eine Speicherzelle und unterstützt die Speicherung und das Abrufen eines einzelnen Datums d. Dies lässt sich ebenfalls auf eine Architektur übertragen, die nach Request-Reply funktioniert. Die angebotenen Methoden in $M_{u,Z}$ sind:

- Lesen: $read(\{\})/[Z \to Z] : d$
- Beschreiben: $store(\{d\})/[Z \to Z'] : \{\texttt{ok}\}$

Die beiden Methoden lassen sich auf die Verben in REST auf diese Weise abbilden (als Beispiel):

- Lesen: $GET_u(\{\})/[Z \to Z] : d$
- Beschreiben: $PUT_u(\{d\})/[Z \to Z'] : \{\texttt{ok}\}$

Beide Anfragen sind idempotent. $read$ oder GET_u liefert das Datum d als Antwort auf die Anfrage zurück. $store$ oder PUT_u nimmt ein beliebiges d entgegen und quittiert den Erfolg den Anfrage. Da $store/PUT_u$ eine schreibende Anfrage ist, ändert sich der Zustand Z in einen Nachfolgerzustand Z'. Gleichzeitig ist $store$ oder PUT_u nicht weiter beschränkt. Ein Datum d, welches im späteren Verlauf ein zweites Mal gespeichert wird, wird ohne Hindernisse in der „Speicherzelle“ abgespeichert, weil es hier keine Definition der Aktualität eines Datums gibt.

Die Fehlerfälle wurden hier aus Gründen der Übersichtlichkeit vernachlässigt. Laut den Vereinbarungen aus 5.4.4, sind Anfragen, die Fehler zurückliefern, safe und damit idempotent. So wird der ressourceninterne Zustand Z nicht beeinflusst.

Der Mechanismus, der hier benutzt wird, erinnert an die Ergebnisse von Lynch und Fischer an [LF81]. Das dort eingeführte RMW-Prinzip („read-modify-write“) kann allerdings in einer agentenorientierten Architektur im allgemeinen nicht atomar (als unteilbare Anweisung) gestaltet werden.

7.1.1 Einführender Hinweis zur Synchronisierung

Bevor erklärt wird wie Synchronisierung gestaltet werden sollte, wird zunächst erklärt, warum nicht jede Art von Synchronisierung in den Architekturen, die hier motiviert wurden, akzeptabel ist. Aus Gegenbeispielen lernt man ebenfalls, deswegen ist es wichtig die Problemstellung einmal zu illustrieren.

$$
\begin{aligned}
&GET_{u_1}/[Z_1 \to Z_1](\{\texttt{If}-\texttt{Modified}-\texttt{Since} : t_{\text{last}}\}) : \texttt{Not modified} \\
&GET_{u_1}/[Z'_1 \to Z'_1](\{\texttt{If}-\texttt{Modified}-\texttt{Since} : t_{\text{last}}\}) : d \\
&PUT_{u_2}/[Z_2 \to Z'_2](\{\texttt{'ack'}\}) : \texttt{ok}
\end{aligned}
\tag{7.1}
$$

In (7.1) ist eine Umgebung (mit M_{u_1,Z_1} und M_{u_2,Z_2} als Methodenmengen im Startzustand) für einen Agenten in einer ressourcenbasierten Architektur beschrieben, in der der Agent zunächst auf eine Änderung des Inhalts einer Ressource mit Referenz u_1 warten muss. Diese Änderung wird von einem Agent durchgeführt, der außerhalb des aktuellen Systems arbeitet und überführt den Zustand Z_1 in Z'_1. Der Agent gibt mit t_{last} den Zeitpunkt der letzten bekannten Änderung an, damit die Ressource das Warten auf die Änderung mitberücksichtigt[1]. Der Agent führt also Polling durch bis ein Datum d zurückgeliefert wird. Dies passiert dann in der zweiten Zeile, wenn M_{u_1,Z'_1} gültig wird. Nach der Änderung

[1] im HTTP-Standard ist so eine konditionale Anfrage sogar im Kopfteil möglich

schickt der Agent eine Antwort und beschreibt dabei Ressource mit Referenz u_2 mit einem `'ack'`, um einem gekoppelten System zu signalisieren, dass der neue Wert d übermittelt worden ist.

Warum ist das spezifizierte Verhalten hier nicht richtig? Man sehe sich dazu das Agentenverhalten im Beispiel (7.1) nochmals an und beobachtet dabei, welche Komponente einen Zustand halten muss, um festzustellen, dass eine Modifikation stattgefunden hat. Es geht hier explizit um die Angabe t_{last}. Hier ist es der *Agent*, der diesen *internen, essentiellen und exklusiven Zustand* speichern muss, damit er feststellen kann, ob eine Änderung bis zum nächsten Aufruf gemacht worden ist. Einen Zustand zu halten ist zwar generell nicht kritisch und Agenten dürfen das im beschränkten Maße tun, aber hier wird der Zustand dauerhaft benutzt, um die Gesamtfunktionalität zu erreichen. t_{last} ist hier essentiell für den resultierenden Gesamtverlauf des Prozesses und wird nirgendwo persistiert. Sollte ein Agent das t_{last} „verlieren“, indem dieser Agent, zum Beispiel, ausfällt, ist nicht mehr feststellbar, ob eine Änderung in der Ressource, die per u_1 referenziert wird, passiert ist, worauf der Agent eigentlich wartet.

Natürlich ist das in einer guten Architektur unerwünscht, dass ein Verlust des Agenten, den gesamten Prozess in einen inkonsistenten Zustand bringen kann und damit den Ablauf des Prozesses ungültig macht. Diese Zustandslosigkeit der Agenten führt dazu, dass Daten auf eine andere Art und Weise ressourcenübergreifend transportiert werden müssen.

Des Weiteren werden in den nächsten Abschnitten stets Programme vorgestellt, die zu jeder Zeit fortführbar sind. Konkret heißt das, dass jeder Agent, der ausfällt das System konsistent belässt, sodass ein späteres Auftauchen des zuständigen Agenten das System weiter in einen korrekt Betrieb versetzen soll. Insbesondere können *alle* Agenten ausfallen und die Systeme verhalten sich beim späteren Betrieb weiterhin korrekt. Das hier verwendete Mittel der *monotonen Sequenzen*, welche ein Gegenstück zu den hier eingeführten *monotonen Ressourcen* darstellen, wird in Kapitel 8.6 detaillierter besprochen.

Man kann den obigen Fall übrigens mit Hilfe von HTTP und konform zum zustandslosen REST auf diese Weise beheben (siehe 7.2), weil HTTP eine Art *test-and-set*-Anweisung unterstützt. Mit dem konditionalen Anweisung im Kopfteil (`If-Unmodified-Since`) wird zuerst überprüft, ob die Ressource ein frischeres Datum enthält.

$$
\begin{aligned}
&GET_{u_1}/[Z_1 \to Z_1]() : \{d, t_{\text{lastmodified}}\} \\
&PUT_{u_2}/[Z_2 \to Z_2'](\{d, \texttt{If-Unmodified-Since:}\, t_{\text{lastmodified}}\}) : \texttt{ok} \\
&PUT_{u_2}/[Z_2 \to Z_2](\{d, \texttt{If-Unmodified-Since:}\, t_{\text{lastmodified}}\}) : \texttt{error}
\end{aligned}
\tag{7.2}
$$

In einem verteilten System wird es allerdings unmöglich die Zeitangabe auf diese Art und Weise zur Synchronisierung zu benutzen, wenn man weiß, dass die Zeit eine globale Ressource ist, die in zwei Ressourcen nicht ohne Abweichungen synchronisierbar ist [Lam78b]. Es muss also auf diesen Mechanismus verzichtet werden. Anstatt von Zeitangaben wird deswegen in diesen Fällen ein lokaler eindeutiger Bezeichner der Ressourcenantwort zugewiesen. Für diese Zwecke hat man bei HTTP den Kopf `ETag` eingeführt [F+99] und erleichtert die Programmierung von Agenten. Es ist darauf zu achten, dass das `ETag` nicht eindeutig für einen Zustand festgelegt ist, sondern für eine eindeutige Antwort. Die Ressource muss zusichern, dass die Antwort exakt gleich ausfällt und nicht durch eine Repräsentation (siehe 6.4.5) oder partielle Antwort (siehe 6.4.7) modifiziert wird. In diesen Fällen muss ein neues `ETag` benutzt werden.

Im Folgenden wird versucht, ohne Seiteneffekte in Agenten, die zur fehlerhaften Berechnung führen könnten, und ohne globale Garantien auszukommen, wie eine konsistente Zeit. Dies ist natürlich eine Voraussetzung für einen agentenorientierten Ansatz.

7.1.2 Sender-Recipient Synchronisierung

Eine generische Ressource ist bereits vielfältig verwendbar für einfachere Synchronisierungsszenarien. Ein solches einfaches Szenario ist die Propagation eines neu aufgetauchten Datums d'.

Gegeben seien zwei Ressourcen in $\mathfrak{R}$ eine Sender-Ressource R_M und eine Recipient-Ressource R_S, mit $u_M \in U(R_M), u_S \in U(R_S)$, und eine Menge von Agenten $G \subseteq \mathfrak{A}$, die das gleiche Programm ausführen. Die Ressourcen sind in Zuständen Z_M und Z_S. Folgende Operationen sind möglich:

$$\begin{aligned} &\text{read}_{u_M}(\{\})/[Z_M \to Z_M] : d' \\ &\text{read}_{u_S}(\{\})/[Z_S \to Z_S] : d \\ &\text{store}_{u_S}(\{d'\})/[Z_S \to Z'_S] : \texttt{ok} \\ &\text{read}_{u_S}(\{\})/[Z'_S \to Z'_S] : d' \end{aligned} \tag{7.3}$$

Nebenbei bemerkt, der Agent, der von R_M den Wert d' ausliest, und diesen ohne Modifikationen in R_S ablegt, führt keine Berechnung durch. Die Funktion für die Übertragung des Datums entspricht der Identitätsfunktion. Hier wurde der einfache Fall zur Veranschaulichung gewählt. Es ist jedoch möglich, und oft wünschenswert, dass der Agent eine Berechnung mit den Eingaben durchführt.

Man setzt hier voraus, dass ein Agent das d' nicht dermaßen verändert, dass ein verfälschtes Datum weiterleitet. Damit schließt man das bösartige und inkorrekte Verhalten der Komponenten aus. Einen authorisierten Zugriff zu steuern ist

generell nicht schwierig (siehe dazu später im Teil 10.7.4) und soll an dieser Stelle nicht erörtert werden.

Was man gut sehen kann, hat das parallele Verhalten der Agentengruppe G in allen möglichen zu bildenden Sequenzen eine eindeutige Gesamtwirkung. Offensichtlich ist nur eine Modifikation (mittels $store$) möglich und auch nur im Ressourcenzustand Z_S. Die Idempotenz sorgt durch das Fehlen der Unterstützung von $store_{u_S}$ dafür, dass der Nachfolgezustand Z'_S erhalten bleibt. Formal ändert sich aber auch nichts, wenn eine Methode $store_{u_S}/[Z'_S \rightarrow Z''_S]$ definiert worden wäre, da angenommen wird, dass nur d' in diesem Beispiel übermittelt werden soll.

7.1.3 Sender-Recipient Synchronisierung mit mehreren Recipient-Komponenten

Die im letzten Absatz eingeführte Sender-Recipient Kommunikation lässt sich auf den Fall verallgemeinern, in dem es mehrere Recipient-Komponenten gibt. Die Synchronisierung ist, wie auch im vorhergehenden Beispiel, ohne das Sperren von Ressourcen möglich.

Das erweiterte Beispiel lautet wie folgt:

$$\begin{aligned} &\text{read}_{u_M}(\{\})/[Z_M \rightarrow Z_M] : d' \\ &\text{read}_{u_{S_i}}(\{\})/[Z_{S_i} \rightarrow Z_{S_i}] : d \\ &\text{store}_{u_{S_i}}(\{d'\})/[Z_{S_i} \rightarrow Z'_{S_i}] : \texttt{ok} \\ &\text{read}_{u_{S_i}}(\{\})/[Z'_{S_i} \rightarrow Z'_{S_i}] : d' \end{aligned} \tag{7.4}$$

Hier wird für jede Recipient-Ressource R_{S_i} mit $i = 1 \ldots n$, wobei n die Anzahl der Recipient-Ressourcen ist, $\text{store}_{u_{S_i}}$ ausgeführt, sodass jede der Recipient-Ressourcen einzeln in den Zielzustand Z'_{S_i} übergeht.

Der Verlauf des gesamten Prozesses der Informationsübertragung führt im Parallelverhalten der gesamten Agentengruppe G stets nur zu einem einzigen Zielzustand der Recipient-Ressourcen.

7.1.4 Beobachtungen bei der Sender-Recipient Übertragung

Bei der Übertragung von Informationen vom Sender auf mehrere Recipient-Ressourcen fällt auf, dass das Überschreiben des Datums in R_M ein Fehlverhalten verursachen kann. Und zwar in der Situation, in der es gewollt ist, dass die

Recipient-Ressourcen jedes Datum „gesehen“ haben, welches zuvor in R_M gespeichert wurde. Dies ist das Verhalten einer typischen Warteschlange mit Garantien bezüglich der Zustellung eines Datums an alle Empfänger [HW03].

Wird hier das Beschreiben von R_M mittels store_{u_M} erlaubt und gilt read_{u_M} in jedem Zustand, den die Ressource annimmt, dann können Informationen wiederholt zu den Ressourcen propagiert werden. Allerdings kann es vorkommen, dass in ungünstigen Fällen eine Recipient-Ressource ein Datum eventuell nicht erhalten hat. Das passiert dann, wenn begonnen wird mit dem Beschreiben von einigen R_{S_i} und der beschreibende Agent ausfällt. Der nächste Agent bekommt unter Umständen schon das nächste Datum und die Recipient-Ressourcen werden in diesem Fall das frühere Datum per $\text{store}_{u_{S_i}}$ nie mitbekommen.

Dieses Problem ist als „Lost-Update“ bekannt [Cas81], das im Abschnitt 7.2 erklärt wird und kann mit generischen Ressourcen ebenfalls, wenn auch sehr umständlich und auch nur zum Teil, gelöst werden. Dies wird im Teil 7.2.1 beschrieben.

7.2 Lost Updates

Ein Problem, bei dem Ordnungen oder Protokolle gefragt sind, sind die sogenannten „Lost Updates“ in verteilten Systemen [Cas81] [BG81]. Diese Situation entsteht wenn, in einer generischen Ressource R zwei Mal hintereinander ein unterschiedliches Datum d' und d'' geschrieben wird, ohne dass es nach dem ersten Schreibzugriff gelesen wurde und die erste Information, also d verwertet wurde. Die folgende Methodenauswahl verdeutlicht das.

$$\begin{aligned}
&\text{read}_u(\{\})/[Z_R \to Z_R] : d \\
&\text{store}_u(\{d'\})/[Z_R \to Z'_R] : \texttt{ok} \\
&\text{read}_u(\{\})/[Z'_R \to Z'_R] : d' \\
&\text{store}_u(\{d''\})/[Z'_R \to Z''_R] : \texttt{ok} \\
&\text{read}_u(\{\})/[Z''_R \to Z''_R] : d''
\end{aligned} \tag{7.5}$$

Bei der folgenden zulässigen Sequenz wurde kein read_u im Zustand Z'_R durchgeführt und ein Eingabewert würde im angekoppelten System nie gesehen werden.

$$\begin{aligned}
&\text{store}_u(\{d'\}); \\
&\text{store}_u(\{d''\}); \\
&\text{read}_u(\{\})
\end{aligned} \tag{7.6}$$

Der Grund ist, dass ein Agent, der sensorisch Ressourcen überwacht, einen Zustand nicht abfragen könnte, weil das Abfrageintervall auf der Ressource R zu groß ist. Allerdings könnte man mit Hilfe einer internen streng monoton steigenden Nummerierung, beim Lesen aus der Ressource, und Dublettenfilterung feststellen, dass ein „Lost Update"-Problem bestanden hat. Um die Erkennung zuverlässig zu gestalten, muss man allerdings nicht nur nacheinanderfolgende Dubletten erkennen, die man mit *idempotenten Methoden* unterdrücken kann, sondern auch verspätete vervielfachte eingehende Nachrichten und dazu dient die *Monotonie*, die in den nächsten Absätzen weiter vorgestellt wird und später mit Hilfe von Datenstrukturen im gewissen Maße realisiert wird.

Die *generische Ressource* kann dieses monotone Verhalten allerdings nicht liefern. Um dies zu realisieren, muss man die Ressourcen mit mehr interner Logik ausstatten. Natürlich darf man in der hier motivierten Architektur das Request-Reply-Paradigma, auf das man hier beschränkt ist, nicht umgehen.

Eine andere Alternative wäre, das Beschreiben der Ressource zu sperren, bis ein Nachweis erbracht worden ist, dass sie gelesen wurde, wie der nachfolgende Teil mit Bestätigungsressourcen (Teil 7.2.1) zeigt.

7.2.1 Bestätigung als Mechanismus für Synchronisierung

Um festzustellen, dass ein Datum alle seine Ziele (Recipient-Ressourcen) erreicht hat, braucht man einen Mechanismus, der mehrere dieser Einzeloperationen ($store$-Aufrufe auf R_{S_i}) zusammenfasst und die Sender-Ressource R_M erst dann wieder freizugeben, wenn das letzte Datum vollständig übertragen wurde.

Ein Weg, um das zu realisieren, ist das übertragene Datum d zu bestätigen, wenn es in einer Zielressource R_{S_i} erfolgreich gespeichert worden ist. Das wird mit einer speziellen Ressource R_{A_i} festgehalten, die als eine Art Bestätigung „acknowledge" fungiert, dass das Datum eines der vorgesehen Ziele erreicht hat. Dazu wird in R_A eine Liste von allen Zielressourcen R_{A_i} mitgeführt. Es wird außerdem angenommen, dass man genügend freie Ressourcen des Typs R_{A_i} instanziieren kann.

Mehrere Agenten führen dazu Methodensequenzen aus und können aus den folgenden Methodenaufrufen in (7.7) wählen.

$$
\begin{aligned}
&\text{read}_{u_M}(\{\}) : \{d\} \\
&\text{read}_{u_A}(\{\}) : \{u_{A_1} \ldots u_{A_n}\} \\
&\text{read}_{u_{A_i}}(\{\})/[Z_i \rightarrow Z_i] : \texttt{false} \\
&\text{store}_{u_{S_i}}(\{d\})/[Z_{S_i} \rightarrow Z'_{S_i}] : \texttt{ok} \\
&\text{store}_{u_{A_i}}(\{true\})/[Z_i \rightarrow Z'_i] : \texttt{ok} \\
&\text{read}_{u_{A_i}}(\{\})/[Z'_i \rightarrow Z'_i] : \texttt{true} \\
&\text{store}_{u_M}(\{d'\}) : \texttt{ok} \\
&\text{store}_{u_A}(\{u'_{A_1} \ldots u'_{A_n}\}) : \texttt{ok}
\end{aligned}
\tag{7.7}
$$

$n + 1$ Agentengruppen können dieses System betreiben. Für jede Zielressource R_{S_i} ist die Gruppe G_i zuständig und eine für das Aktualisieren von R_M, die Gruppe G_M. Für die letztere Agentengruppe lässt sich das folgende Programm (7.8) vereinbaren. Es wird `RESTART` benutzt, um zu veranlassen, dass der Agent die Arbeit an der Stelle abbricht und zum späteren Zeitpunkt wieder starten kann.

$$
\begin{aligned}
&\text{read}_{u_A}(\{\}) : \{u_{A_1} \ldots u_{A_n}\}, \\
\exists i \in 1, \ldots, n : &\texttt{if } \text{read}_{u_{A_i}}(\{\}) : \texttt{false, then RESTART} \\
&\text{store}_{u_M}(\{d'\}) : \texttt{ok}, \\
&\text{store}_{u_A}(\{u'_{A_1} \ldots u'_{A_n}\}) : \texttt{ok} \\
&\texttt{RESTART}
\end{aligned}
\tag{7.8}
$$

Während jeder Agent in Gruppe G_i das folgende Programm (7.9) ausführt.

$$
\begin{aligned}
&\text{read}_{u_M}(\{\}) : \{d\}, \\
&\text{read}_{u_A}(\{\}) : \{u_{A_1} \ldots u_{A_n}\}, \\
&\texttt{if } \text{read}_{u_{A_i}}(\{\}) : \texttt{true, then RESTART} \\
&\text{store}_{u_{S_i}}(\{d\}) : \texttt{ok}, \\
&\text{store}_{u_{A_i}}(\{true\}) : \texttt{ok} \\
&\texttt{RESTART}
\end{aligned}
\tag{7.9}
$$

Es ist zu bemerken, dass die Sender-Ressource R_M nun mehrmals mit einem frischerem Datum (zum Beispiel oben d') beschrieben werden kann, um es von einzelnem Nachrichtentransport (siehe 7.1.3) abzugrenzen. Die Strategie, die hier

identifiziert werden kann, basiert auf dem pessimistischen Sperren [Dij71]. Logisch gesehen, wird R_M für das Beschreiben gesperrt, solange alle R_{S_i} den Wert noch nicht mitgeteilt bekommen haben. Dieses Beispiel ist etwas einfacher und erfordert nicht, dass Systeme, die an R_{S_i} gekoppelt sind, den Erhalt bestätigen. Das kann dazu führen, dass R_{S_i} durch das nachfolgende Datum überschrieben wird, bevor das angekoppelte System das Datum konsumiert. Das Problem des „Lost Update“ ist also nur lokal im aktuellen System gelöst und dazu mit relativ viel Aufwand.

Die Beobachtung hier ist, dass die Ressourcen R_{A_i} implizit für Kontrolle gebraucht werden und für Monotonie auf der Ordnung der (lokalen) Nachrichtenaktualität sorgen. Dies reicht jedoch nicht aus, um zu behaupten, dass R_M Monotonie bewahrt. Der Agent, der R_M beschreibt, kann, nachdem alle Agenten die Übertragung bestätigt haben, einen älteren Wert d^* in Ressource R_M hineinschreiben. Hier gibt es jedoch auch keine Informationen über die Aktualität von d^*, an denen man die Verletzung der Monotoniebedingungen feststellen könnte.

Außerdem ist auffällig, dass das Vorgehen in diesem Übertragungssystem stärker synchronisiert ist. Es wird jeweils genau ein Datum an alle abhängigen Systeme weiter gegeben und das nächste Datum angefordert, wenn alle diese Systeme versorgt wurden. Hier wird also versucht die Bedingung der *Atomizität* [HR83] zu erfüllen, indem ein konkurrenzausschließender Mechanismus ähnlich der einer *Transaktion* [GR92] [LBK01] benutzt wird.

Was man aus diesem pessimistischen Sperren von Ressourcen mitnehmen sollte ist insgesamt also folgendes:

- keine Garantien für Monotonie
- schlechte Parallelisierung wegen Sperrung in R_M
- die Sperrungen als Maßnahme gegen das „Lost Update“-Problem pflanzen sich in angekoppelten Systemen fort (an den R_{S_i}) und synchronisieren sehr stark

Es ist beachtenswert, dass zur Steuerung dieser einfacher Synchronisierungsmaßnahmen lediglich *generische Ressourcen* zum Einsatz gekommen sind. Diese beinhalten keine erweiterte Logik, außer der Speicherung. Wenn man jedoch die Ressource R_A aus den obigen Beispielen näher anschaut, wird zum ersten Mal klar, dass hier eigentlich eine Datenstruktur zur Hilfe genommen wird, um die Kontrolle zu unterstützen. Es werden nämlich *URI*s in einer *Menge* zusammengefasst, um den gesamten Ablauf zu kontrollieren. Des Weiteren ist es zu beachten, dass die Kontrollressourcen R_{A_i} und R_A getrennt von den datenspeichernden Ressourcen selbst sind, R_M und R_{S_i}. Dies wurde im Teil 3.12 motiviert.

7.3 Funktionsweise der generischen Ressourcen

Mit der generischen Ressource ist es möglich, Prozesskontrolle im beschränkten Maße auszuüben. Es ist nämlich gezeigt worden, wie Informationen von einer Ressource auf eine oder mehrere andere Ressourcen zu übertragen sind, um die Ausbreitung von Information und zu unterstützen. Im Hinblick auf die Prozesswelt, ist es schon mit der generischen Ressource möglich, Daten zwischen Systemen auszutauschen und sogar signalisieren, dass ein Datum an mehrere Systeme weitergereicht worden ist.

Möchte man die Informationsausbreitung kompositional betrachten, muss man sich die Funktionsweise der Agenten genauer anschauen. Überträgt man ein Datum von einer Ressource auf eine andere, kann das durch einen Agenten erkannt werden, der die Zielressource im Definitionsbereich hat. Er kann mit diesem Verfahren zur Arbeit *aktiviert* werden, weil eine Veränderung festgestellt werden kann. Hierzu ist es nötig, Informationen über den Zustand Z der Ressource zu haben, wie er vorher war. Es ist nicht nötig alle Daten über den Ursprungszustand vorzuhalten. Es reicht, die Steuerungsinformationen zu betrachten, zum Beispiel eine eindeutig zugeordnete Ressource, die den Prozesszustand für das angekoppelte System festhält, wie es im Teil 7.2.1 vorgeführt wurde. Auf diese Weise können bereits übermittelte Daten bestätigt und der Gesamtprozess kann weiter fortgeführt werden.

Diese Art von Informationsübertragung ist jedoch nicht mächtig genug, um *kooperative Prozesse* verteilt und mit spärlich eingesetzter *Synchronisierung* zu unterstützen. Es wäre ein größerer Vorteil für die Entkopplung in einem Architekturstil, das *optimistische Sperren* zu unterstützen, damit die Arbeit in angekoppelten Systemen nicht aufgehalten wird und das natürliche Prinzip der entkoppelten Arbeit in der Architektur beibehalten wird.

7.4 Kommunikationskanäle auf Ressourcen

Eine interessante Möglichkeit zur Informationsübertragung beschreibt das Verfahren namens *HTTPLR* [dh05], welches inoffiziell als ein Internet „Draft“ veröffentlicht wurde und für einiges an Begeisterung in der REST-Gemeinde gesorgt hat.

Das dort eingeführte Protokoll und die Ressourcen lösen das Problem ein Datum *genau ein Mal* und zwar zuverlässig zwischen Server und Client (also Ressource und Agent) zu übertragen. Um das Problem zu lösen, wird ein *Kommunikationskanal* mit Hilfe einer Ressource erstellt, der nach der Übertragung geschlossen wird.

Konkret funktioniert die Übertragung folgendermaßen:

1. Kanal-Ressource instanziieren und URI anfordern
2. Datum in Kanal speichern
3. Datum von Kanal abholen
4. Kanal-Ressource vernichten

Das Verfahren nutzt einen kontrollierten Nebeneffekt der Instanziierung im System, der ausdrücklich im Teil 5.4.2 erlaubt wurde.

HTTPLR war eine Inspiration für die Erweiterung der Kommunikation mit der sich diese Arbeit beschäftigt. Die Frage, wie man in einem echten verteilten System kontrolliert Informationen überträgt geht noch weiter als HTTPLR mit der zuverlässigen Server-to-Client-Übertragung (und umgekehrt) demonstriert. In Architekturen, die kooperative Prozesse unterstützen, beschäftigt man sich mit dem Problem, Informationen zu Berechnungen zuzuordnen und mit Reihenfolgen zu einem Gesamtergebnis zu kommen. Zudem geht es darum, die Information zuverlässig zwischen den Ressourcen zu bewegen und nicht nur zwischen Agenten und Ressourcen.

7.5 Datenstrukturen für Prozesskontrolle

Um Prozesskontrolle zu unterstützen, müssen Ressourcen Reihenfolgen und Ordnungen beachten. Man muss die Identität eines Datums feststellen können und ein einzelnes Datum mit einem anderen vergleichen können. Der erste Schritt in diese Richtung stellt eine Ressource, die mit einer Datenstruktur angereichert ist. Das alles scheint zunächst einfach zu sein, aber wenn man bedenkt, dass die Ordnung mit Hilfe von lokalen Eigenschaften rekonstruiert werden muss, bekommt das Problem eine neue Facette. Verteilte Architekturen sollten auf globale Systemeigenschaften verzichten. Die Ressourcen in der ressourcenbasierten Architektur haben keine Möglichkeiten globale Systemeigenschaften zu vereinbaren, weil sie miteinander nicht kommunizieren können.

Wie schon im Teil 7.3 erwähnt, können Eingaben mit Hilfe von generischen Ressourcen zur Kopplung an weitere Tätigkeiten im System gespeichert werden. Die Problematik dabei ist, dass auch identifiziert werden muss ob die Eingaben, die üblicherweise aus verteilter Bearbeitung stammen, tatsächlich durch alleinige Änderung der Eingabe, einen angekoppelten Prozess anstarten sollen.

Das erste Problem ist die Behandlung von Ordnungen auf Daten. Die generischen Ressourcen können durch *Idempotenz* feststellen, dass ein unmittelbar

nachfolgendes Datum (zum Beispiel, eine weitere gleiche Lösung eines Teilproblems) nicht weiter betrachtet werden muss. Das Problem entsteht, wenn solch eine redundante Lösung in Sequenz als Nachfolger nach anderen aktuelleren Lösungen an das gekoppelte System übergeben wird. An dieser Stelle hilft Idempotenz nicht weiter und es müssen andere Mittel angewendet werden.

Ein kurzes Beispiel für so eine Sequenz ist (7.10). Hier gibt es eine Diskrepanz zwischen den geschriebenen Daten d und d' und den zugeordneten Zuständen Z' und Z'''. Nun besteht die Möglichkeit, dass das Datum d verspätet aus einer parallelen Berechnung stammen kann. Ein Prozess welches dieses Datum d verspätet erhält, stellt nur den neuen Zustand Z''' fest und würde an dieser Stelle, wo die Ankopplung stattfindet, wegen der Zustandslosigkeit der Agenten, eine solche Wiederholung nicht erkennen können.

$$\begin{aligned} &\text{store}(d)/[Z \to Z'] : \{\texttt{ok}\}; \\ &\text{store}(d')/[Z' \to Z''] : \{\texttt{ok}\}; \\ &\text{store}(d)/[Z'' \to Z'''] : \{\texttt{ok}\} \end{aligned} \tag{7.10}$$

Es wird klar, dass hier ein zusätzliches Mittel in Frage kommt, das im Teil 5.4.5 als *Monotonie* vorgestellt worden ist. Wenn man eine Funktion t definieren kann, die die *Aktualität* des Datums feststellen kann, ist man auch in der Lage, die resultierenden redundanten Berechnungen in verschiedenen Zuständen zu unterdrücken. *Monotonie* wirkt hier sozusagen, ähnlich wie *Idempotenz*, nur über Zustände hinaus, die die Ressource zwischenzeitlich angenommen hat.

Im Folgenden werden Datenstrukturen vorgestellt, die mit Hilfe von bestimmten Idempotenz- und Monotonieeigenschaften einen Einfluss auf Kontrolle in Prozessen ausüben können. Zuerst jedoch wird der Aufbau einer Ressource vorgestellt, die die Kontrolle mit Hilfe einer internen Datenstruktur übernimmt.

7.6 Aufbau von datenstrukturengetriebenen Ressourcen

Das Konzept der datenstrukturgetriebenen Ressource zur Prozessansteuerung ist im Falle von REST nicht so neu. Man findet im Web zahlreiche Beispiele, die Atom-Feeds oder RSS/RDF dazu nutzen, um Aufgabenlisten zu organisieren und eine Ordnung in die Ausführung bringen sollen. Bei einer solchen Aufgabenliste handelt sich, von der Intuition her, eine Aneinanderreihung von Ressourcen, die mit Hilfe von *URI*s referenziert werden und von Agenten gleich behandelt werden sollen.

Diese Idee ist von der Datenorganisation hervorragend geeignet, um Kontrolle in Prozessen auszudrücken und dabei die eigentlichen Daten aus dem Kontrollkonzept auszulagern in externe Ressourcen. Damit erkennt man hier zum wiederholten Male das Konzept der *Trennung von Daten und Kontrolle* und die daraus resultierenden Vorteile, die im Teil 3.12 benannt worden sind. Zudem wird einer generischen Lösung gesucht, um solche Kontrolle ausüben zu können.

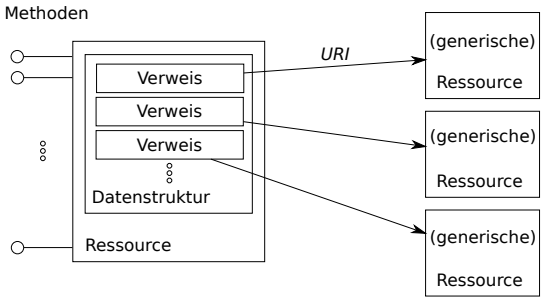

Abbildung 7.1: Ressource mit Datenstruktur

7.7 Warteschlangen

Das klassische Beispiel einer Datenstruktur, welche Daten in Geschäftsprozessen transportiert ist die Warteschlange [HW03]. Warteschlangenimplementierungen realisieren in BPM-Systemen einen Nachrichtentransportmechanismus, der Persistenz unterstützt, gewisse wählbare Zustellgarantien erfüllt und synchron arbeitet. Synchron ist eine Warteschlange deswegen, weil ein eingereihtes Datum typischerweise mit Hilfe von Subscription-Mechanismen weiter propagiert wird. Diese, wie auch jede andere Synchronisierungsart, wirkt störend auf die Verteilungseigenschaften und die Entkopplung.

Für den ressourcenbasierten Architekturstil, ist es nötig, dass eine Warteschlange in datenbasierter Form vorliegt, als passives Element. Der Antrieb muss alleine durch das Request-Reply-Paradigma erfüllt werden. In Kooperation mit redundant geschalteten Agenten ist zudem die Erkennung von vervielfachten Informationen nützlich, sodass Zustandsmodifikationen auf einfache Weise eingespart werden.

Durch die Entkopplung soll es möglich sein, Eingabe für weitere Systeme in der Form anzubieten, dass diese nicht per Push-Prinzip, sondern per Pull-Prinzip [Eag85] bedient werden können. Insbesondere bei kooperativen Prozessen ist es wichtig, weil die Laufzeit von einem Nachfolgersystem nicht verlangt wird, sondern völlig unabhängig *zur Verfügung gestellt* wird.

7.7.1 Pull-Prinzip und dessen Konsequenzen

Einer der Nachteile einer verteilten und entkoppelten Architektur ist, dass das Verfahren *Publish-Subscribe* [HW03] nicht gut umsetzbar ist. Es ist nämlich nicht möglich, ein Verfahren nach dem *Push-Prinzip* zu realisieren. Das Request-Reply-Pardigma, welches der Architektur zu Grunde liegt, macht es unmöglich, eine synchronisierte Abarbeitung umzusetzen. Dieses Problem wird später im Teil 10.7.1 angesprochen.

Die Interaktion von Agenten mit den Ressourcen ist deswegen nach dem *Pull-Prinzip* gestaltet. Ein Verfahren wie Publish-Subscribe kann jedoch als eine passive Variante betrieben werden. Bevor man diesen Mechanismus vorstellen kann, muss vorher die Funktionsweise der *Warteschlangen* (Teil 7.7.2) erklärt und die Modifikation mit dem Ziel der *Multiinstanzfähigkeit* eingeführt werden (siehe Teil 7.9.3).

7.7.2 Funktion einer Warteschlange in der Architektur

Um Information koordiniert in einer ressourcenbasierten Architektur bewegen zu können, muss man gewisse Steuerungsmechanismen zu Hand haben. Es ist nämlich nicht immer der Fall, dass man Information vervielfältigt, sondern muss diese auch an bestimmten Stellen konzentrieren und zusammenfassen.

Da Ressourcen und die darin gespeicherten Daten inaktiv sind und Interaktion mit anderen Ressourcen stets Agentenaktionen erfordert, möchte man aus Effizienzgründen das lokale Verhalten einer Datenstruktur dazu benutzen, um Steuerung in der ressourcenbasierten Architektur zu ermöglichen.

Für die Verarbeitung von Information sind oft geordnete Reihenfolgen interessant. Eine Warteschlange ist im Prinzip eine ganz einfache Datenstruktur, die als ein Behälter für Information dienen kann. Der gesamte Inhalt der Warteschlange unterliegt einer Ein- und Ausgabeordnung.

Die klassische Warteschlange als abstrakte Datenstruktur funktioniert nach dem Prinzip „first-in-first-out“ (FIFO). Es gibt typischerweise zwei Methoden, die diese Funktionalität realisieren:

`queue(E):` hänge ein Element am Ende der Warteschlange an (Eingabeseite)

`E:=unqueue():` hole ein Element von Anfang der Warteschlange, lösche es und gib es zurück (Ausgabeseite)

Dies könnte schon eine gültige Ressource in $\mathfrak{R}$ darstellen. Um das Paradigma der Parallelisierung aufrecht zu erhalten und Agenten beliebig redundant anzusehen, muss die Ausgabeseite in zwei separate Schritte geteilt werden. Mit einem

klassischen `unqueue` wäre das Element, welches durch einen Ausfall nicht bearbeitet worden ist, endgültig verloren. Dies ist eine unerwünschte Eigenschaft, welche zu Informationsverlust führen würde. Die Warteschlange wird deswegen, entsprechend der Anforderungen in der Architektur, angepasst (siehe dazu: Abbildung 7.2). Die `unqueue`-Anweisung wird deswegen in zwei Einzelmethoden aufgetrennt. Das nächste Element in der Ausgabe soll angeschaut werden können mit `peek`. Erst wenn die Bearbeitung durch einen *Konsumenten* auf der Ausgabeseite abgeschlossen ist, wird das Element aus der Warteschlange mit `delete` entfernt. Es ist wichtig, dass beide Methoden *idempotent* sind und dass `delete` diese Methodensequenz abschließt, die Agenten auf der Ausgabeseite durchführen. Das Verhalten der Agenten und der Begriff der *monotonen Sequenz*, die hier vorliegt, wird im nächsten Kapitel 8 ausführlich besprochen.

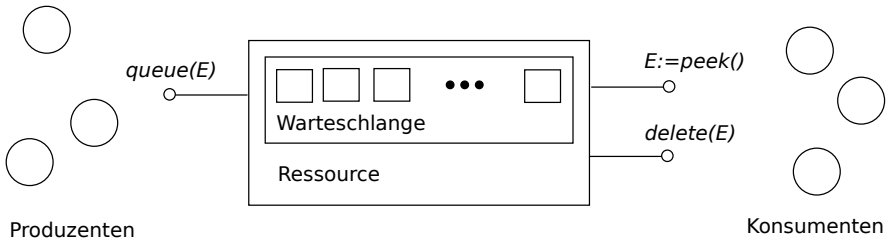

Abbildung 7.2: Ressource mit darunterliegenden Warteschlange als Kontrollstruktur

Die Eingabeseite der Warteschlange setzt keine *Idempotenz* voraus. Die Methode `queue` ist vom Charakter her nicht idempotent, hat aber in bestimmten Varianten durch Monotonieeigenschaften auf den Ordnungen, die im Nachfolgenden untersucht werden, idempotente Eigenschaften. Im Gegensatz zur Ausgabeseite kann, mit Hilfe von Ordnungen, auf zusätzliche externe Kontrollfunktionalität für eingehende Elemente verzichtet werden.

Der Effekt ist, dass die Methodenaufrufe am Eingang der Warteschlange zu beliebigem Zeitpunkt in einer Methodensequenz passieren können. Dieser Effekt ist später genauer erklärt, wenn das Agentenverhalten genauer analysiert wird. Er hilft monotone Sequenzen aufzubauen, bei denen die Eingabeseite zwar den Ressourcenzustand modifiziert, aber im Parallelverhalten Nebenwirkungen unterdrückt, die aus wiederholten Aufrufen, zum Beispiel innerhalb einer Agentengruppe, stammen könnten.

Ankommende Elemente, die in der Warteschlange gespeichert werden, werden passiv auf ihre Merkmale untersucht, um Ordnungen festzustellen. Dazu sind lokale Kriterien geeignet, die nun besprochen werden. Die Ordnungen in den Warteschlangen sind auf der Basis von *monotonen Methoden* entworfen worden, die die Warteschlangen zur Verfügung stellen. Solche Art von Methoden reduzieren die Komplexität des Entwurfs nach dem ressourcenbasierten Architekturstil.

7.7.3 Lokale Ordnungen einer Warteschlange

An dieser Stelle ist zuerst zu klären welche lokalen Eigenschaften kann man sich zu Nutze machen, wenn man eine Warteschlange R mit Referenz $u \in U(R)$ als Datenstruktur hat. Die Warteschlange nutzt verschiedene Ordnungen auf den ankommenden Paketen. Diese Ordnungen müssen im zugehörigem Zustand Z lokal gespeichert werden.

Drei Ordnungen, die ohne besonderen Rechenaufwand zu ermitteln sind, sind die folgenden:

interne Nummerierung (N): die ankommende Elementnummer wird inkrementiert, jedes Mal wenn ein Element angehängt wird

Nummer des frühesten äquivalenten Vorkommens (T): man muss dazu die ankommenden Elemente vergleichen können, ob sie zu der gleichen Berechnung gehören und vergibt eine neue Nummer, nur dann wenn das ankommende Element noch nicht gesehen wurde, sonst wird die früheste bekannte Nummer verwendet

Ordnung auf Elementwerten (V): dazu muss man eine Ordnung, direkt auf den Werten der Elemente definiert haben, um sie vergleichen zu können, das ankommende Paket hat sozusagen einen vorbestimmten Platz in der Warteschlange, solange es nicht angekommen ist

Zur Illustration der Ordnungen wird an dieser Stelle ein Beispiel gegeben. Es wird angenommen, dass die Elementenreihe mit den Werten $\{1, 1, 2, 4, 3, 4, 3, 4\}$ in die Warteschlange nacheinander eingegeben wird mit Hilfe der Methode $\text{queue}_u(p)$. Der Wert ist direkt durch den formalen Parameter p angegeben, also kann dieser einfach mit der Identitätsfunktion berechnet werden. Der Vergleich auf T wird ebenfalls zur Vereinfachung direkt auf den Werten durchgeführt. Dies führt, für die drei oben genannten Ordnungen, zu den Reihen in der Tabelle in Abbildung 7.3.

Elementnummer	**N**	1	2	3	4	5	6	7	8
Wert	**V**	1	1	2	4	3	4	3	4
Vorkommen	**T**	1	1	2	3	4	3	4	3

Abbildung 7.3: Lokale Nummerierung von ankommenden Elementen

Auf der Eingabeseite der Warteschlange gibt es nun ein Mittel, um ankommende Elemente mit *Monotonie* oder *strenger Monotonie* abzuspeichern. Dazu wird die abkürzende Schreibweise *I* für englisch „Input“, dann optional *S* für „strictly“ und eine der obigen drei Ordnungen benutzt, also *V*, *N* und *T*. Da alle diese

Ordnungen mindestens monotone Eigenschaften haben, würde hier ein weiteres Symbolbuchstabe für „Monotonie“ redundant sein. Als Beispiel ist *ISN* die *strenge Monotonie auf der internen Nummerierung der Elemente auf der Eingabeseite der Warteschlange*.

Auf der Ausgabeseite einer Warteschlange gibt es generell zwei Arten von Verhaltensweisen für die Elemente, die durch die Eingangseigenschaften „vorsortiert“ wurden. Die erste Art hat keine Ausgabeordnung und modelliert den wahlfreien Zugriff auf den Warteschlangenelementen. Die zweite Art ist die strenge Monotonie, die sich in den meisten Fällen auf *N* (interne Elementnummerierung) bezieht. Eine weitere Alternative ist *V* (für Elementwerte). Ordnungen auf *T* auf der Ausgabeseite werden hier nicht gebraucht. Als Beispiel ist *OSN* die *strenge Monotonie auf der internen Nummerierung der Elemente auf der Ausgabeseite der Warteschlange*.

Um das vollständige Verhalten solch einer monotoniegetriebenen Warteschlange zu beschreiben, wird die vollständige Bezeichnung mit einem Schrägstrich verbunden. Zum Beispiel ist *ISN/OSN* eine gültige Warteschlangenbezeichnung.

Im folgenden werden die unterschiedlichen *Warteschlangentypen* angegeben, die nach den obigen Prinzipien konstruiert werden können. Die meisten benötigen eine weitere Datenstruktur, um die Ordnung möglichst effizient zu berechnen. Es fällt dabei auf, dass die internen Datenstrukturen die allgemein bekannten einfachen abstrakten Datenstrukturen sind. Sie reichen aus, um das gesuchte Verhalten zu realisieren.

Es ist wichtig zu bemerken, dass die Warteschlangenstrukturen hier ein *generisches Modell* zur Konstruktion sind. Sie beschreiben das *ressourceninterne Verhalten* und geben keine exakte Implementation an. Die resultierende Implementation weicht oft ab, wegen bestimmter erwünschter Eigenschaften, die der Programmierer der resultierenden Ressource gerne hätte. Eine solche Implementation, die die Warteschlange realisiert, lässt sich auf eines der genannten Verhalten abbilden und die Steuerung in der internen Implementation lässt sich trotzdem generisch auf die aufgeführten Typen festlegen.

7.8 Monotone Warteschlangen

Warteschlangen kommen zum Einsatz bei der Ankopplung von Systemen, die Daten in einer festgelegten Reihenfolge liefern. Die verschiedenen Arten von Ordnungen werden mittels *Monotonie* und strenger Monotonie auf den zugehörigen unterstützenden Datenstrukturen angeboten.

Da Warteschlangen in der Architektur mit Verweisen auf Elemente (*URI*s) arbei-

ten, ist es nötig, die erforderlichen Angaben, die zur lokalen Bestimmung der Ordnung auf den Elementen nötig sind, direkt in der Hauptstruktur zu speichern. Aus Gründen der *Trennung von Kontrolle und Daten*, kann es dazu kommen, dass URIs auf Systeme verweisen, deren Elementdaten von der Kontrollressource aus nicht ohne weitere Operationen zugreifbar sind (Ressourcen können selbst keine Zugriffe auf andere Ressourcen außerhalb von Z machen). Dabei kann die Ordnung unter Umständen nicht berechnet werden. Dieses Problem ergibt sich vor allem bei dem Ordnungskriterium V. Man sollte hier so vorgehen, dass man möglichst präzise die Funktion t bestimmt (siehe 5.10 im Teil 5.4.5), welches für die vorliegenden Elemente in der Warteschlange gilt. Auf diese Weise werden die entscheidenden Kontrollinformationen extrahiert und die eigentlichen Elementdaten können erst bei Bedarf, von Agenten auf der Ausgabeseite, in einer separaten Leseoperation angefordert werden.

Zur Veranschaulichung des Verhaltens der spezifischen Warteschlangen, wird nochmals die Folge aus der Tabelle in Abbildung 7.3 im Teil 7.7.3 benutzt. Dabei symbolisieren die eckigen Klammern (`[...]`; Feldklammern) die geordnete Ausgabe und die geschweiften Klammern (`{...}`; Mengenklammern), die wahlfreie Ausgabe. Um die Reihenfolgen besser nachzuvollziehen, wird auf das Entfernen von Elementen aus der Ausgabe verzichtet (delete$_u$). Was man also beim Punkt **Beispielfolge** sieht, ist der Inhalt der Warteschlange nach den Einfügeoperationen. Im Falle der geordneten Ausgabe, wird auf das Element zugegriffen, das am weitesten links steht. Und so lange das Element nicht entfernt wird, kann nicht mit der Bearbeitung des nächsten gestartet werden (konsequenterweise wird es also nur möglich sein, das erste Element zu entfernen). Die geordneten Warteschlangentypen erlauben auch eine geschlossene Folge von Elementen auszulesen, die links stehen. Man sollte aber stets daran denken, dass das Entfernen von Elementen so gestaltet werden muss, dass die Warteschlange mit Überschneidungen von Elementenintervallen zurecht kommen muss, was zusätzliche Komplexität in Form eines erhöhten Implementierungsaufwands bedeutet. Bei dem wahlfreien Zugriff kann hingegen jedes Element in beliebiger Reihenfolge entfernt werden, was sehr vielfältige Kombinationen der Befüllungen der Warteschlange erlaubt.

7.8.1 *ISN/OSN*

In Worten:	streng monotone Eingabe auf interner Nummerierung, streng monotone Ausgabe auf interner Nummerierung
Ordnungsrelation:	$<_N$
Verwendung:	klassischer *FIFO*, der die Eingangsreihenfolge beibehält
Beispiel:	Aufgabenliste, die mit Bevorzugung von Redundanz abgearbeitet wird
Datenstrukturen:	Warteschlange
Beispielfolge:	`[1,1,2,4,3,4,3,4]`

Diese Warteschlange speichert alle Elemente in der gleichen Reihenfolge, in der sie empfangen wurden. Mehrere Agenten auf der Eingabeseite können die gleichen Informationen in der Struktur einreihen. Diese werden dann mehrfach abgespeichert. Auf der Ausgabeseite wird jeder Agent auf dem Element arbeiten, welches am längsten in der Warteschlange wartet und noch nicht gelöscht worden ist. Dies führt zur Bevorzugung einer redundanten Bearbeitung der einzelnen Elemente.

7.8.2 *ISN/ON*

In Worten:	streng monotone Eingabe auf interner Nummerierung, wahlfreier Zugriff anhand interner Nummerierung
Ordnungsrelation:	$<_N$
Verwendung:	vorsortierte Eingabe, alle eingegebenen Werte werden zur Verfügung gestellt
Beispiel:	Aufgabenliste, die mit Bevorzugung von Verteilung abgearbeitet wird
Datenstrukturen:	Warteschlange
Beispielfolge:	`{1,1,2,4,3,4,3,4}`

In dieser Warteschlange werden wie auch bei *ISN/OSN* in der Reihenfolge gespeichert, in der sie empfangen wurden und es gelten die gleichen Eigenschaften auf der Eingangsseite wie bei *ISN/OSN*. Die Ausgabeseite ist hingegen anders gestaltet. Sie betont die parallele Arbeitsweise und bevorzugt die Verteilung der Arbeitskraft auf den Elementen. Die Agenten auf der Ausgabeseite können die Reihenfolge der Elemente nachvollziehen, haben aber dennoch die freie Wahl, welches Element sie wählen und später, nach der Bearbeitung, entfernen. Da hier

ein Nichtdeterminismus der Wahl des Elements auf die Seite des Agenten verlagert wird, kann die Arbeitsweise hier flexibel bezüglich der Abarbeitungsreihenfolge gestaltet werden.

7.8.3 *IV/OSN*

In Worten:	monotone Eingabe auf Werten, streng monotone Ausgabe auf interner Nummerierung
Ordnungsrelation:	$\leq_V$
Verwendung:	es werden nur Werte weiter gereicht, für die gilt: $t(v_{\text{neu}}) \geq t(v_{\text{letztes}})$
Beispiel:	gibt Werte weiter, die das Maximum erreichen oder übersteigen in chronologischer Ordnung
Datenstrukturen:	Warteschlange und $t(v_{\text{letztes}})$
Beispielfolge:	`[1,1,2,4,4,4]`

Mit Hilfe dieser Art von Warteschlange, ist es möglich nach der Aktualität der Daten selbst zu unterscheiden. Die Eingabe, die nach dem Kriterium V filtert und vorsortiert, sichert zu, dass ein, laut t, älteres Datum nicht eingereiht wird. Natürlich ist eine Operation auf V aufwändiger als auf T. Während es bei T lediglich um Feststellung der Gleichheit eines Datums geht und dazu reicht die Untersuchung der Referenz (*URI*) aus, muss bei V das Datum selbst an die Kontrollstruktur übergeben werden, um t anzuwenden und damit die Ordnung festzustellen. Die Ausgabeseite bevorzugt wieder die Redundanz in Agentengruppen.

Insgesamt verhält sich die Warteschlange so, dass ältere Nachrichten (laut t) als diese, die aktuell empfangen wurde, nicht beachtet werden und mittels der monotonieerzwingenden Methode zum Anhängen an die Warteschlange, einfach ignoriert werden. Das aktuelle Maximum bezüglich t wird jedoch weiter gereicht. Dieses Verhalten kann in Prozessen verwendet werden, in denen verspätete Informationen zuvor ausgefiltert werden sollen, bevor sie an weitere Systeme gereicht werden.

7.8.4 *IV/ON*

In Worten:	monotone Eingabe auf Werten, wahlfreier Zugriff anhand interner Nummerierung
Ordnungsrelation:	$\leq_V$
Verwendung:	vorsortierte Eingabe auf Werten, für die gilt: $t(v_{\text{neu}}) \geq t(v_{\text{letztes}})$
Beispiel:	stellt die ganze Historie von Werten zur Verfügung, die mindestens den maximalen Stand erreicht haben
Datenstrukturen:	Warteschlange und $t(v_{\text{letztes}})$
Beispielfolge:	`{1,1,2,4,4,4}`

Dieser Warteschlangetyp unterscheidet sich von der Funktion des Typs *IV/OSN*, lediglich dadurch, dass auf der Ausgabeseite die freie Wahl der noch nicht bearbeiteten Elemente besteht. Da aber auch das „frischeste“ Element von der Warteschlange entfernt werden könnte, ist darauf zu achten, dass der Maximalwert von t separat gespeichert wird, damit die Monotonie bezüglich V aufrecht erhalten werden kann.

Die Ausgabeseite kann vielfältig genutzt werden und kann natürlich auch mehrere Werte auswählen und löschen, da sie zwar logisch anhand von Werten geordnet sind, aber nicht geordnet angeboten werden müssen. Damit wird wieder die Möglichkeit zur parallelen Bearbeitung eröffnet.

7.8.5 *ISV/OSN*

In Worten:	streng monotone Eingabe auf Werten, streng monotone Ausgabe auf interner Nummerierung
Ordnungsrelation:	$<_V$
Verwendung:	es werden nur Werte weiter gereicht, für die gilt: $t(v_{\text{neu}}) > t(v_{\text{letztes}})$
Beispiel:	gibt neue Maximumwerte in chronologischer Reihenfolge weiter
Datenstrukturen:	Warteschlange und $t(v_{\text{letztes}})$
Beispielfolge:	`[1,2,4]`

Die Warteschlange *ISV/OSN* verhält sich fast wie *IV/OSN*, mit dem Unterschied, dass nur die Werte eingereiht werden können, die echt größer laut t sind. Der Einsatz dieses Warteschlangentyps erlaubt es, Werte oder Nachrichten weiter zu reichen, die auf jeden Fall aktueller sind als alles was zuvor empfangen wurde.

Die Ausgabeseite koppelt die folgenden Agenten redundant an.

7.8.6 *ISV/ON*

In Worten:	streng monotone Eingabe auf Werten, wahlfreier Zugriff anhand interner Nummerierung
Ordnungsrelation:	$<_V$
Verwendung:	vorsortierte Eingabe auf Werten, für die gilt: $t(v_{\text{neu}}) > t(v_{\text{letztes}})$
Beispiel:	stellt die ganze Historie von Werten zur Verfügung, die ein neues Maximum erreicht haben
Datenstrukturen:	Warteschlange und $t(v_{\text{letztes}})$
Beispielfolge:	`{1,2,4}`

Wie auch *ISV/OSN* (siehe 7.8.5) reiht diese Warteschlange nur neue Nachrichten ein, aber erlaubt wahlfreien Zugriff auf der Ausgabeseite und bietet damit die Möglichkeit zur parallelen Bearbeitung der Elemente. Da eine Ordnung auf V benutzt wird, muss auch hier der Maximalwert laut t gepuffert werden, da das neueste Element vorzeitig gelöscht werden könnte.

7.8.7 *IST/OSN*

In Worten:	streng monotone Eingabe auf dem Vorkommen, streng monotone Ausgabe auf interner Nummerierung
Ordnungsrelation:	$<_T$
Verwendung:	Vermeidung von redundanten Folgen von Ergebnissen und Sortierung nach dem ersten Vorkommen zur Weiterbearbeitung
Beispiel:	akzeptiert das erste Vorkommen eines Resultats einer Berechnung als gültiges Element
Datenstrukturen:	Warteschlange und Hash
Beispielfolge:	`[1,2,4,3]`

IST/OSN erlaubt Ergebnisse redundanter Berechnungen aus den angekoppelten Systemen auszufiltern und somit anschließende Berechnungen nur einfach anstatt vielfach auszuführen. Diese Warteschlange reiht also nur Elemente ein, die zum ersten Mal aufgetreten sind und gibt sie chronologisch, bezüglich der Reihenfolge des ersten Auftretens weiter.

Der Warteschlangentyp *IST/OSN* ist eine spezielle Form des Warteschlangentyps *ISV/OSN* (siehe 7.8.5). Während *ISV/OSN* eine Abbildung t benötigt, um Äquivalenz auf V zu entscheiden, ist bei der Nummerierung des letzten Vorkommens

mittels T, nur die Zuordnung der empfangenen Daten zu einer bestimmten Berechnung wichtig. Es ist deswegen eine Verallgemeinerung, weil man die Berechnungszugehörigkeit im Datum selbst angeben kann und t diesbezüglich entsprechend vereinbart.

7.8.8 *IST/ON*

In Worten:	streng monotone Eingabe auf dem Vorkommen, wahlfreier Zugriff auf interner Nummerierung
Ordnungsrelation:	$<_T$
Verwendung:	Vermeidung von redundanten Folgen von Ergebnissen
Beispiel:	akzeptiert das erste Vorkommen eines Resultats einer Berechnung als gültiges Element
Datenstrukturen:	Warteschlange und Hash
Beispielfolge:	`{1,2,4,3}`

Die Eingabeseite des Wartenschlangentyps *IST/ON* funktioniert wie die von *IST/OSN* (siehe 7.8.7). Die Ausgabeseite erlaubt den wahlfreien Zugriff auf die eingereihten Elemente, was zur parallelen Bearbeitung gebraucht wird.

Der Warteschlangentyp *IST/ON* verhält sich zu *ISV/ON*, wie *IST/OSN* zu *ISV/OSN*. Auch hier handelt es sich um die entsprechende Spezialisierung, die lediglich die T-Reihenfolge benötigt und keine Abbildung t.

Die Besonderheit dieser Warteschlange ist, dass sie Elemente, die bereits gesehen worden sind, nicht ein zweites Mal weiterreicht und eine freie Auswahl der Elemente, die noch nicht bearbeitet worden sind, anbietet. Diese Warteschlange verhält sich also wie ein Hash-basierter Speicher, der sich tatsächlich als ein sehr nützliches Werkzeug erweist und der deswegen weiter im Teil 7.11 untersucht wird.

7.8.9 *IT/OSN*

In Worten:	monotone Eingabe auf dem Vorkommen, streng monotone Ausgabe auf interner Nummerierung
Ordnungsrelation:	$\leq_T$
Verwendung:	Filterung älterer verspäteter Ergebnisse mit Wiederholung aktueller Vervielfachungen und geordnete Ausgabe
Beispiel:	-
Datenstrukturen:	Warteschlange, Hash und letztes Element
Beispielfolge:	`[1,1,2,4,3,3]`

Löst man die strenge Monotonie auf T auf der Eingabeseite auf, dann erhält man ein Verhalten, das etwas ungewöhnlicher ist. Der Warteschlangentyp *IT/OSN* lässt zu, dass Wiederholungen noch nicht gesehener Elemente in die Warteschlange eingereiht werden können.

Die Ausgabeseite reicht die durch die Eingabeseite zugelassenen Elemente in der N-Reihenfolge an ein angekoppeltes System weiter. Wie bei *IST/OSN* kommt bei diesem Warteschlangentyp ebenfalls ein Hash-basierter Speicher für die interne Verwaltung zum Einsatz.

Bei der Implementation dieses Warteschlangentyps ist darauf zu achten, dass die Warteschlange entleert werden könnte und die Information verloren gehen könnte, welches Element zuletzt empfangen wurde, um über eine zulässige Wiederholung zu entscheiden. Deswegen sollte man das zuletzt empfangene Element noch einmal separat speichern.

7.8.10 *IT/ON*

In Worten:	monotone Eingabe auf dem Vorkommen, wahlfreier Zugriff auf interner Nummerierung
Ordnungsrelation:	$\leq_T$
Verwendung:	Filterung älterer verspäteter Ergebnisse mit Wiederholung aktueller Vervielfachungen
Beispiel:	-
Datenstrukturen:	Warteschlange, Hash und letztes Element
Beispielfolge:	`{1,1,2,4,3,3}`

Der Wartenschlangentyp *IT/ON* lässt ebenfalls zu, dass einkommende noch nicht vorgekommende Elemente auf der Eingabeseite vervielfacht gespeichert werden

können. Jedoch erlaubt die Ausgabeseite den wahlfreien Zugriff auf Elemente, die zu jeder Zeit entfernt werden können.

Bei der Implementation dieses Warteschlangentyps ist darauf zu achten, dass die Sequenz der zuletzt eingereihten Elemente gelöscht werden könnte, deswegen ist es nötig, das letzte gesehene Element zusätzlich zu speichern, um zu entscheiden, ob es sich um eine zulässige Wiederholung handelt oder nicht.

7.8.11 *IST/OSV*

In Worten:	streng monotone Eingabe auf dem Vorkommen, streng monotone Ausgabe auf Werten
Ordnungsrelation:	$<_V$
Verwendung:	vollständige Wertesortierung, mit konsekutiven Zugriff
Beispiel:	Weiterleitung von bereits gesehenen Elementen mit vollständiger Sortierung
Datenstrukturen:	Warteschlange und Array
Beispielfolge:	`[1,2,3,4]`

Ein interessanter Warteschlangentyp ist *IST/OSV*. Das Gesamtverhalten sorgt nämlich für eine totale Ordnung auf dem V-Kriterium. Dies ist nur möglich, wenn der Wertebereich der Elemente vollständig bekannt ist und außerdem gilt, dass jedes Element in der Ordnung irgendwann eingereiht wird.

Die Eingabeseite sorgt dafür, dass sich nicht wiederholenden Elemente in die interne Datenstruktur eingereiht werden. Die Ausgabeseite hat ein ganz besonderes Verhalten. Bis jetzt wurden Warteschlangentypen nur mit dem N-Kriterium für die Ausgabeseite gebraucht. Das V-Kriterium sorgt dafür, dass die Ausgabeseite nur dann Elemente liefert, wenn es Elemente gibt, die lückenlos aufsteigend auf ihren V-Werten gesehen worden sind.

Das bedeutet, dass wenn in der Beispielfolge oben die 4 gespeichert worden ist, diese so lange nicht ausgeliefert wird, bis die Werte $1, \ldots, 3$ ebenfalls abgespeichert worden sind.

Intern lässt sich dieses Verhalten mit einer Array-Struktur modellieren. Dieses Array ist anhand von Indizes passend für die Position der Werte aufgebaut. Der Entwickler muss im Gegensatz zu den anderen Warteschlangentypen genau wissen, welche Werte erwartet werden. Das „Wann“ bleibt hier allerdings offen.

7.8.12 *IST/OV*

In Worten:	streng monotone Eingabe auf dem Vorkommen, wahlfreier Zugriff auf Werten
Ordnungsrelation:	$<_V$
Verwendung:	vollständige Wertesortierung, mit wahlfreien Zugriff auf konsekutiven Werten
Beispiel:	wahlfreie Weiterleitung von bereits gesehenen Elementen mit vollständiger interner Sortierung
Datenstrukturen:	Warteschlange und Array
Beispielfolge:	`{1,2,3,4}`

Wie der Warteschlangentyp *IST/OSV*, ist *IST/OV* von der Eingabeseite exakt gleich aufgebaut. Es werden ebenfalls Elemente eingereiht, die vom Wert her noch nicht vorgekommen sind.

Der einzige Unterschied liegt im Verhalten der Ausgabeseite. Diese erlaubt den wahlfreien Zugriff auf Elemente, die bereits eingereiht worden sind, noch nicht aus der Warteschlange entfernt worden sind und ein geschlossenes Intervall bezüglich des Kriteriums V bilden.

In der Beispielfolge oben bedeutet das, dass wenn beispielsweise die 1 gelesen und entfernt worden ist und die nächsten Elemente in der Reihenfolge $4, 3, 2$ eingetroffen sind, dann stehen anschließend alle diese drei Elemente zur Wahl beim auslesen. Der Entwickler dieses Warteschlangentyps muss hier unterscheiden zwischen einem Element, welches noch nicht gesehen worden ist und einem das bereits entfernt wurde.

7.8.13 Sonstige Warteschlangentypen

Die hier aufgeführten Warteschlangentypen bilden die grundlegenden Verhaltensweisen bei der Übergabe von Informationen zwischen zwei Systemen nach.

Die Auswahl der sinnvollen Kombinationen auf der Eingabeseite besteht aus den Eingabeverhalten $\{ISN, IV, ISV, IST, IT\}$. Das Eingabeverhalten *IN* gibt es nicht, weil die Nummerierung der Eingaben stets streng monoton ist. Demgegenüber steht das Ausgabeverhalten $\{OSN, ON, OSV, OV\}$.

Es entstehen noch folgende Kombinationsmöglichkeiten, die übrig sind.

IT/OSV entspricht eingangsseitig dem Verhalten von *IT/OSN* 7.8.9. Zusammen mit der strenger Monotonie auf V wird die Ausgabeseite ein instabiles Verhalten an den Tag legen. Es liegt daran, dass das letzte Vorkommen laut T nicht den höchsten Wert laut V haben muss und damit ist es möglich, dass

kleinere Werte in der Warteschlange eingangsseitig akzeptiert werden und größere Werte gleichzeitig entfernt werden.

IT/OV würde ein ebenfalls instabiles Verhalten abhängig von der Ausgabeseite auf die Eingabeseite übertragen. Dies ist unerwünscht, wie im Falle von *IT/OSV*.

ISN/OSV* und *ISN/OV sind ineffizientere Varianten der Warteschlangentypen *IST/OSV* 7.8.11 und *IST/OV* 7.8.12. Sie sind vom Verhalten jedoch gleich.

IV/OSV*, *IV/OV*, *ISV/OSV* und *ISV/OV sind ebenfalls ineffiziente Entsprechungen der Warteschlangentypen *IV/OSN* 7.8.3, *IV/ON* 7.8.4, *ISV/OSN* 7.8.5 und *ISV/ON* 7.8.6.

Mit „Ineffizienz“ ist hierbei die umständliche Form der Implementation eines solchen Kandidanten für einen Warteschlangentypen gemeint, wobei dieser das gleiche Verhalten wie die aufgeführte Entsprechung aufweisen würde. Damit sind alle anderen Kandidaten bezüglich der Kriterien N, T und V ausgeschlossen. Andere lokale Kriterien für Ordnungen wurden hier nicht betrachtet.

7.9 Modifikationen der monotonen Warteschlangen

Im Teil 7.8 wurden Warteschlangentypen vorgestellt, die keine konkrete Implementationen sind, sondern vielmehr ein Prinzip beschreiben, nachdem die Monotonie und die Ordnung der Elemente verwaltet wird.

Es wurden jeweils paarweise zwei Ausführungen eines Warteschlangentyps erwähnt. Einer, der geordnete Ausgabe anbietet und strenge Monotonie aufweist, welche zur Bevorzugung des redundanten Verhaltens von Agenten führt und ein weiterer Typ, der wahlfreie Ausgabe benutzt, um den Verteilungsaspekt zu betonen. Die beiden Arten des Ausgabeverhaltens der Elemente an die Agenten stehen zueinander in Konflikt und können nicht beide gleichzeitig optimiert werden.

Es ist auch zu beachten, dass bei der strengen Monotonie, die Wahl des zu ausliefernden Elements intern durch die Warteschlange entschieden wird. In der Begriffswelt der Web-Dienste (als Ressourcen) modelliert dies die *interne Wahl*. Hier entscheidet die Ressource intern anhand der benutzten Ordnung auf den Elementen. Im Falle des wahlfreien Zugriffs obliegt die Wahl des Elements dem Agenten und stellt sich als *externe Wahl* dar. Die beiden Begriffe der Wahlarten wurden von Hoare im Zusammenhang mit CSP eingeführt [Hoa78].

7.9.1 Paginierung

Einen Kompromiss zwischen den beiden Extrema stellt die *Paginierung* des Elementeangebots dar. Mit dieser Art der Modifikation der Ausgabeseite wird lediglich eine beschränktes Interval auf der Ausgabeseite zur Verfügung gestellt. Man modifiziert dazu einen Warteschlangentyp mit wahlfreiem Angebot auf der Ausgabeseite, eine feste oder adaptiv gestaltete Zahl von Elementen anzubieten. Falls mehr Elemente zur Verfügung stehen, werden diese erst dann angeboten, wenn Elemente im aktuellen Angebot abgearbeitet werden. Der Ausdruck *Paginierung* entstammt aus der Präsentation von Suchresultaten von Web-Diensten, bei denen man seitenweise die nächsten Treffer in Form einer längeren Liste angezeigt bekommt. Warteschlangen sind ebenfalls in Listenform darstellbar und können so implementiert werden, dass sie Intervalle zurückliefern, die entweder die Warteschlange anbietet oder der Agent selbst auswählt.

Bei der Implementierung der Paginierung ist darauf zu achten, dass im gleichen Ressourcenzustand stets das gleiche Angebot zur Auswahl vorliegt. Eine willkürliche Wahl würde die *Zustandslosigkeit* der Ressource verletzen, die im Teil 5.4.1 vereinbart wurde.

7.9.2 Blockierende Eingabe

Alle genannten Warteschlangentypen mit strenger Monotonie auf der Ausgabeseite können so implementiert werden, dass sie nur maximal ein aktives Element speichern und damit auf die interne Datenstruktur der Warteschlange verzichten. Die andere interne Datenstruktur (wie Hash und Array) bleibt jedoch erhalten. Elemente, die temporär nicht in der Warteschlange eingereiht werden können, werden zunächst abgewiesen, bis die Warteschlange entleert wird. Natürlich ist diese Betriebsart ineffizient, aber sie kann dazu genutzt werden, um die Eingabeseite stärker zu synchronisieren. Zum Beispiel kann im Falle des Warteschlangentyps *IST/OSV* erzwungen werden, dass die Agenten, die einkommende Elemente abspeichern, diese in der richtigen Reihenfolge einreihen. Dies führt dazu, dass das Verhalten der Agenten auf der Eingangsseite strenger kontrolliert werden kann und kann genutzt werden, um den Effekt der zu realisieren.

Diese Art von Betrieb ist aus offensichtlichen Gründen für die Warteschlangentypen mit wahlfreiem Zugriff uninteressant.

7.9.3 Multiinstanzausgabe

Eine weitere mögliche Modifikation stellt die *Multiinstanzfähigkeit* dar. Diese gestaltet sich so, dass mehrere Agentengruppen die Elemente der jeweiligen Warte-

schlange mehrfach abrufen können und die Warteschlange für die Agentengruppen verschiedene Zustände abspeichert.

Normalerweise funktioniert die Warteschlange im Eininstanzbetrieb so, dass ein Element global für alle Agenten in $\mathfrak{A}$ entfernt wird. Es kann jedoch interessant sein, die Warteschlange so zu verwalten, dass die Übertragung zu verschiedenen Zielsystemen separat und unabhängig voneinander verwaltet wird.

Das Verfahren ähnelt im Grunde „Publish-Subscribe“, nur erfordert es nicht dass Agenten oder Zielsysteme explizit eine Subscription haben. Um dies zu realisieren, modifiziert man für die Warteschlange an der *URI* q die Methode $\texttt{delete}_q$ folgendermaßen.

$$\texttt{delete}_q(\{\text{Id}(d), \text{dest}_i\}) \tag{7.11}$$

Von der Semantik her hat dieses modifizierte $\texttt{delete}_q$ einen zusätzlichen Parameter dest_i bekommen, anhand welchen festgestellt wird, *für welches Zielsystem* das Datum d anhand dessen Identifikation (sei sie hier durch die Funktion $\text{Id}(d)$ gegeben) als abgearbeitet markiert werden soll.

Ein erheblicher Nachteil dieser Warteschlange ist, dass sie nie entleert werden kann, weil sie durch fehlendes Wissen über die angebundene Systemzahl nicht entscheiden kann, ob das erste eingereihte Datum bereits entfernt werden könnte.

Die Multiinstanzausgabe ist jedoch zustandsfrei. Ein Agent einer Agentgruppe kennt nämlich stets sein Ziel dest_i oder kann das anhand von anderen Ressourcen entsprechend ermitteln. Auf diese Weise ist abgesichert, dass der Agent nicht essentielle, private Zustandsinformationen hält.

7.9.4 Entkoppelte Alternative zu „Publish-Subscribe“

Um *Publish-Subscribe* zu realisieren muss man, wie schon im Teil 7.7.1 angemerkt, auf das *Push-Verfahren* verzichten. Unterstützend zu einem Warteschlangentyp kann man die Ressource so modifizieren, dass eine Methode $\texttt{subscribe}_q(\text{dest}_i)$ eingesetzt wird, die das Kriterium N (interne Nummerierung) hinzuzieht und damit feststellt zu welchem Zeitpunkt ein Zielsystem die Warteschlange abonniert hat. Die Entnahme der Elemente funktioniert dann weiter exakt wie bei der Multiinstanzausgabe im Teil 7.9.3 und aktualisiert nebenbei den Status bezogen auf das jeweilige Zielsystem.

Im Unterschied zur Multiinstanzausgabe können hier jedoch Aufräummechanismen eingesetzt werden. Wenn das letzte Zielsystem ein Element aus der Warteschlange entnimmt, kann dieses Element freigegeben werden. Das liegt daran,

dass ein neuer Aufruf von `subscribe` niemals eine schon vergebene Nummer anhand des Kriteriums N bekommen kann.

Bei Warteschlangentypen mit strenger Monotonie auf der Ausgabeseite kann die Freigabe auch relativ einfach durch die Bestimmung des Minimus auf den zugehörigen N-Werten der jeweiligen Zielsysteme geschehen. Im anderen Falle gestaltet sich der Aufräumvorgang durch die freie Wahl der Elemente etwas aufwändiger. Es müssen nämlich alle Zielsysteme ein Element entnommen haben (mit `delete`). Für jedes eingefügte Element ist hier dann ein Feld (zum Beispiel als Vektor) mitzuführen, welches die Entnahme für ein einzelnes Zielsystem an der jeweiligen Position im Feld festhält. Ist an allen Positionen im Feld des Element als gelöscht markiert worden, kann das Element dann aus der Warteschlangenressource gelöscht werden, weil jedes Zielsystem die Löschung angefordert hat.

7.10 Ressourcen zum Synchronisieren rekursiver Probleme

Für *rekursiv definierte Probleme*, ist aus der Sicht eines Informatikers natürlich der Keller als abstrakte Datenstruktur eine geeignete Form, um Kontrolle zu verwalten. Im Falle von Architekturen, in denen parallele Abarbeitung mittels unbestimmter Zahl von Agenten geschieht, ist es nicht möglich den Keller so einzusetzen wie man es von herkömmlichen Szenarien kennt.

Um den Keller anzupassen, braucht man zu allererst die Monotonie nach der Definition in 5.4.5, die bei der rekursiven und parallelen Abarbeitung von Teilaufgaben, nicht zu ihrer Vervielfachung führt. Aus diesem Grund werden die üblichen Methoden auf der abstrakten Datenstruktur zum Teil modifiziert.

`E:=peek():` da diese Methode nur lesend und damit *safe* (und konsequenterweise *idempotent*) ist, kann man die ursprüngliche Semantik des `peek()` auf einem Keller beibehalten.

`pop*(E):` diese Methode löscht das angegebene Element E, wobei die ursprüngliche Löschung sich ausschließlich auf das oberste Element des Kellers bezieht und deswegen ohne die Angabe von E auskommt.

`push*(E):` mit dieser Methode wird das Element in der Ressource im Keller abgelegt. Sie unterscheidet sich von der ursprünglichen Kellermethode dadurch, dass Elemente bei Identität nicht vervielfacht werden. Sollte ein Element Im Keller bereits vorhanden sein, wird das ältere Element (also das, welches nicht oben liegt) entfernt. Sollte es nicht mehr im Keller gespeichert sein, weil es mittels `pop*` gelöscht wurde, wird das Element nicht gespeichert und der Zustand der Ressource unverändert belassen.

Es ist hierbei zu beachten, dass die hier konstruierten Methoden `push*` und `pop*` den Monotoniebedingungen vollkommen genügen.

Der auf diese Weise veränderte Keller ist natürlich im eigentlichen Sinne kein Keller mehr, weil die Ordnung der Elemente bei den modifizierenden Methoden nicht eingehalten wird. Untersucht man das Ressourcenverhalten, stellt man aber fest, dass diese Ressource dem Warteschlangentyp *IST/OSN* entspricht (7.8.7). Um die Äquivalenz herzustellen, muss man die Nummerierung auf N dekrementierend gestalten. Die Ordnung auf T sorgt für Monotonie und stellt sicher, dass die (hier rekursive) Berechnung bei der Nutzung dieser Datenstruktur nicht in alte Zustände zurückfällt.

7.11 Assoziative Ressourcen

Wie auch die anderen Ressourcen basiert der Charakter der *assoziativen Ressource* auf einer internen Datenstruktur. Es handelt sich hierbei um ein assoziatives Feld (Hashtabelle). Dabei haben die eingegebenen Elemente ihre vorgesehenen Plätze und können stets genau ein Mal gespeichert werden. Die assoziative Ressource wird deswegen gerne für das abschließende Zusammenfügen des Gesamtproblems anhand von Teillösungen in einem kooperativen Prozess gebraucht.

Analysiert man das Verhalten weiter, stellt sich heraus, dass diese sehr stark an den Warteschlangentyp *IST/ON* 7.8.8 angelehnt ist. Diese Ähnlichkeit ist nicht zufällig, sondern reflektiert exakt das Verhalten des Hashes. Nicht zuletzt liegt es daran, dass auch dort der Hash als interne Datenstruktur gebraucht wird und die Präferenz, durch die einfache Monotonie auf der Ausgabeseite, auf der Verteilung liegt.

Um eine assoziative Ressource vielseitiger verwenden zu können, wird diese auch durch weitere Methoden ergänzt, um Fragen nach der *Existenz* des Elements in der Datenstruktur stellen zu können.

7.12 Entkopplung von lastintensiven Berechnungen

In heutigen Systemen wird zunehmend auf die sofortige Konsistenz verzichtet, um lastintensive Berechnungen nicht direkt in den Ressourcen durchzuführen. Die Ressourcen sollten möglichst zeitnah eine Antwort auf eine Anfrage beim Methodenaufruf generieren, weil das Request-Reply-Paradigma eingehalten werden sollte.

Eine Möglichkeit, um solche eventuelle Konsistenz zu erreichen, die schon im Teil 3.12 angesprochen worden ist, ist, die benötigten Parameter des Methoden-

aufrufs zwischenzuspeichern und die eigentliche Arbeit von einem angekoppelten System erledigen zu lassen. Dies führt dazu, dass der Effekt des Methodenaufrufs nicht sofort in Erscheinung tritt, weil das angekoppelte System den Methodenaufruf zunächst bearbeiten und später entsprechend synchronisieren muss.

Die Abbildung 7.4 beschreibt diesen Zusammenhang. Die beiden Agenten A_1 und A_2 gehören verschiedenen Systemen in einer verteilten ressourcenbasierten Architektur an. Die ressource in der Mitte sorgt für Laufzeitentkopplung der beiden Systeme. Der Agent A_1 übergibt einen Parameter p an die Datenstruktur, wo dieser gepuffert wird und im angekoppelten System weiter verarbeitet wird. Der Agent bekommt dann auch die Antwort „Accepted“, worauf der Agent die zusätzliche Information bekommt, dass der Aufruf entkoppelt und deswegen verspätet bearbeitet wird. Natürlich muss das zweite System am Ende der Bearbeitung das Element als abgearbeitet markieren. Hier wird es durch den Aufruf der Methode `delete` realisiert. Auf die Details bezüglich des Verhaltens der Agenten wird später im Teil 8 eingegangen.

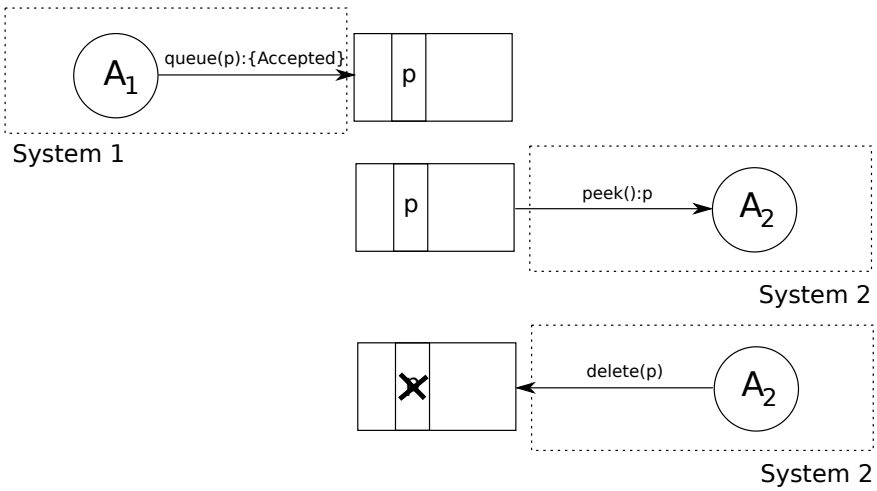

Abbildung 7.4: Entkopplung der Laufzeit zweier Systeme mit Hilfe einer Ressource

In der Welt des Webs wurde im Protokoll *HTTP* dazu eine spezielle Antwort auf eine Anfrage reserviert, die keine sofortige Wirkung zeigen muss. Es handelt sich um den Antwortstatus „202 Accepted“, der im Abschnitt 10.2.3 der HTTP-Spezifikation [F+99] beschrieben worden ist.

Solch ein Mechanismus unterstützt die Entkopplung von Systemen in einer Architektur. Ein System welches einen Methodenaufruf durchgeführt hat, kann unabhängig vom Bearbeitungszustand zunächst ohne Verzögerungen weiter arbeiten. Der Effekt der Anfrage kann durch ein, von der Laufzeit her, völlig entkoppeltes System, erzeugt werden.

Für diese Arbeit eignen sich Warteschlangen verschiedener Typen sehr gut. Welche davon in Frage kommt, hängt von der Arbeitsweise des angekoppelten Systems ab. Zum Beispiel können durch monotone Methoden der Warteschlangen nach 7.8 in einem kooperativen Prozess redundante Eingaben direkt durch die Ressource herausgefiltert werden. Hierzu eignet sich das Kriterium T am Eingang der Warteschlange.

7.13 Abschließende Bemerkungen zu Ressourcen

In diesem Kapitel wurden zunächst *generische Ressourcen* (im Teil 7.1) vorgestellt. In einer ressourcenbasierten Architektur sind diese Art von Ressourcen minimal ausgestattet. Es wurde untersucht wie mächtig diese einfachen Datenspeicher sind und wo deren Grenzen sind.

Durch das Verhalten der Agenten in Agentengruppen, wurde eine Ressourcenart gesucht, deren Eigenschaften den Agenten helfen redundante Ergebnisse zu kontrollieren (siehe Teil 7.5). Es wurde auch gezeigt, dass pessimistische Sperrungen nicht für die Lösung des „Lost-Update"-Problems geeignet sind (Teil 7.2), weil sie viel zu radikal auf eine verteilte Architektur einwirken.

Die Vermeidung von redundanter Ausführung zusammen mit der nichtsperrenden Art und Weise zur Behebung des „Lost-Update"-Problems führte dazu, dass monotone Methoden mit einer unterliegenden Datenstruktur gefunden worden sind (Teil 7.7.3). Es wurde zunächst die Warteschlange als solch ein Datentyp vorgeschlagen, da die Ordnungen auf lokalen Kriterien dazu ausreichen, viele Arten von Synchronisierungen zu realisieren. Das Verhalten der Warteschlange wurde in verschiedene Warteschlangentypen gegliedert (im Teil 7.8). Anschließend wurde gezeigt, dass einige andere interne Ressourcendatenstrukturen sich auf die Warteschlangentypen abbilden lassen (Teile 7.10 und 7.11). Zugleich wurden Modifikationen von Warteschlangen vorgestellt (Teil 7.9), die für einige Szenarien bei der Informationsausbreitung unter den Systemen nützlich und möglich sind.

Die Idee der Entkopplung im Teil 7.12 zeigt wie man Systeme mit Hilfe der konstruierten Ressourcen komponieren kann und zwar so, dass die Entkopplung bewahrt wird, die in der untersuchten ressourcenbasierten Architektur erhaltenswert ist.

In den Teilen 7.1.1 und 7.7.2 wurden bereits *monotone Sequenzen* erwähnt, aber noch außenvor gelassen, was sie für eine Bedeutung haben. Das nächste Kapitel 8 zeigt, dass das Verhalten der Agenten bestimmten Prinzipien folgen muss, um Informationen in konsistenter Form an Systeme zu übergeben. Erst das Agentenverhalten und die datenstrukturgetriebenen Ressourcen ergeben zusammen

die Eigenschaften, die nötig sind, Informationsweitergabe zwischen Systemen in einer vorrangig datenbasierten Architektur zuverlässig zu gestalten, die lediglich das Request-Reply-Prinzip kennt.

Für die Organisation von Agentengruppen mit ihrem redundanten Verhalten sind besonders Warteschlangentypen mit Eingabeverhalten IST und ISV interessant. Sie erweisen sich in der Praxis als die nützlichsten Varianten bei parallelisierten Bearbeitung von verteilten Aufgaben und kooperativen Prozessen. Sieht man sich unter den Merkmalen die „Beispielfolge“ an, dann erkennt man, dass diese die einzigen Warteschlangentypen sind, die *Idempotenz* unterstützen (siehe Teil 5.5), die für diese Zwecke notwendig ist.

Kapitel 8

Agentenverhalten

Im Kapitel 7 standen Ressourcen ($\mathfrak{R}$) im Fokus der Betrachtung. Das Verhalten im System wird jedoch auch von Agenten ($\mathfrak{A}$) und ihrer Organisation spezifiziert.

Das Verhalten der Agenten, und explizit die Art und Weise wie Methodenaufrufe passieren, führt zu einigen interessanten Beobachtungen. In einem System, das auf Sperrungen aus Gründen besserer Parallelisierung verzichten soll, müssen einige Regeln eingehalten werden. In den nachfolgenden Teilen werden diese Regeln und ihre Eigenschaften näher untersucht und präzisiert. Im Vordergrund dieser Untersuchung stehen Kriterien zur Protokollkonstruktion. Ein solches Protokoll ist die Beschreibung des *lokalen Verhaltens* eines Agenten. In einer verteilten Architektur, welche Agenten erlaubt unabhängig voneinander zu arbeiten, gibt es keine vorgegebenen Mechanismen, das Verhalten von zwei unterschiedlichen Agenten mit einer und der gleichen synchronisierten Ablaufordnung zu beschreiben und so einzugrenzen, wie man es von Workflows her kennt.

Die Herausforderung dabei ist, das Verhalten lokal zu beschreiben mit einem *Agentenprogramm* und dabei die Ressourcen zu nutzen, um einen Einfluss auf alle anderen beteiligten Agenten auszuüben. Bei der Beschreibung der Ressourcen waren zwei Eigenschaften daran beteiligt, um die Auswirkungen des Verhaltens innerhalb von Agentengruppen etwas einzugrenzen. Das waren einerseits die Idempotenz (im Teil 5.4.3) und die Monotonie (Teil 5.4.5). In diesem Kapitel werden diese Mittel weiter verwendet, um Agentenprogramme zu entwickeln, die eine konsistente Form von Informationsübertragung zwischen verschiedenen Systemen innerhalb einer ressourcenbasierten Architektur realisieren.

Die Konstruktionshinweise zu Ressourcen aus dem letzten Kapitel und die Prinzipien, die hier vorgestellt werden, ergeben zusammen eine Möglichkeit, in einer ressourcenbasierten Architektur, kooperative Prozesse (Teil 4) zu gestalten, die aus Systemen bestehen, die mit Hilfe von Daten kommunizieren und dadurch entkoppelt arbeiten können.

8.1 Kontrolle und Konsistenz

Im Teil 3.12 wurde die Trennung von Kontrolle und Daten motiviert. Für diesen Zweck wurden URIs eingeführt, um Daten mit Hilfe von Referenzen zu adressieren und so diese für andere Zwecke von anderen Kontrollressourcen aus zu gebrauchen. Daten sollten aus Effizienzgründen länger unveränderlich bestehen bleiben als die Zustandsdaten, die für Kontrolle gebraucht werden. Die Referenzierung per *URI* wird hier auch *Indirektion* genannt, weil sie im gewissen Sinne der indirekten Adressierung eines Prozessors ähnelt, der im ersten Schritt die Adresse nachschlägt und anhand dieser erst das eigentliche Datum. Natürlich kann man das Verfahren der Indirektion sowohl für einfache Daten als auch Daten, die Kontrollzustände reflektieren verwenden.

Vollzieht man eine Trennung von Daten und Kontrolle, dann wird das Problem, die Konsistenz zu wahren vor allem auf den Kontrollanteil der Daten verlagert. Das führt dazu, dass für die Kontrolle oftmals Ressourcen (datenstrukturgetrieben) gebraucht werden sollten, wie diese, die im Teil über Warteschlangentypen erwähnt worden sind (siehe 7.8). Für Daten kommen einfache generische Ressourcen in Frage, die sogar bei Bedarf erstellt werden können.

Allgemein gilt, dass die Konsistenz der Kontrolle dann gegeben ist, wenn ein vollständiger Ablauf eines Agentenprogramms die Datenwelt in den nächsten Zustand befördert, mit dem wiederum ein Agentenprogramm weiter arbeiten kann. Konsistenz muss hier in zwei verschiedenen Formen betrachtet werden.

1. Innerhalb einer Agentengruppe dürfen beliebige Präfixe der Sequenzen von Methodenaufrufen in Kooperation mit mindestens einem erfolgreichen Programmablauf nicht zu verschiedenen Ergebnissen führen.
2. Produziert eine Agentengruppe G_1 eine Ausgabe, die von einer anderen Agentengruppe G_2 weiter verarbeitet werden soll, muss diese Ausgabe so in die Ausgaberessourcen geschrieben werden, dass die Agentengruppe G_2 sie als eine Gesamtheit nach Abschluss der Arbeit von G_1 erkennt.

Die erste Konsistenzform beschäftigt sich also mit der Konsistenz innerhalb einer Agentengruppe und die zweite mit der Konsistenz der Datenwelt im Falle der Kooperation von zwei oder mehr angekoppelten Agentengruppen.

Bei der ersten Form von Konsistenz geht es um Präfixe von Sequenzen. In der ressourcenbasierten Architektur ist wegen des Request-Reply-Mechanismus sehr leicht festzustellen, ob eine Methode fehlgeschlagen ist. Ein Agent bricht an dieser Stelle ab und das Präfix endet dort. Genau das gleiche passiert im Falle eines Ausfalls eines Agenten. Es entsteht ebenfalls ein Präfix einer Sequenz. Da ein

Agent zustandslos ist und anderen Agenten keine Information vorenthält, die essentiell für die Fortführung wäre, ist es völlig irrelevant welcher Agent eine Sequenz erfolgreich beendet.

Im Laufe dieses Kapitels wird man feststellen können, dass erst die Zusammenwirkung der monotonen Datenstrukturen zusammen mit einer besonderen Form von Sequenzen von Methodenaufrufen diese beiden Konsistenzformen hier erfüllt.

8.2 „Einfache“ Synchronisierung mit Mashups

Eine einfache Form der Synchronisierung, die fast schon als diese gar nicht betrachtet wird, ist ein Mashup, das sehr oft im Web eingesetzt wird. Hierbei geht es darum, die Inhalte einer Web-Seite mit externen Inhalten zu erweitern, die im aktuellen Kontext zur aktuellen Web-Seitenansicht stehen. Ein Beispiel wäre die berühmte Standortanzeige zu einer publizierten Hausadresse anhand einer Karte, die von einem externen Anbieter kommt. Ein solcher Dienst wird von der Firma *Google* in Form der *Google Maps* [Goo11] Web-Applikation angeboten. Auf dieser Weise wird oft in Adressbüchern, Firmenregistern und ähnlichen, zu einer gewählten Adresse auch eine kleine Karte miteingeblendet, wo der Standort markiert ist. Dabei muss die überliegende Seite, bei einer Änderung ihres Kontextes, diese Änderung in der externen Web-Anwendung mitreflektieren.

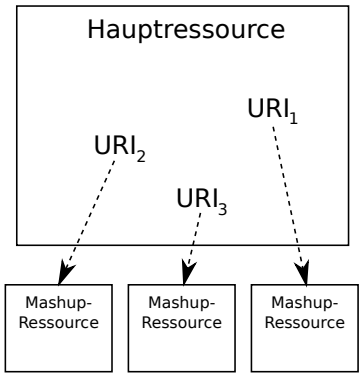

Abbildung 8.1: Mashups

Hier werden offensichtlich Zustände sehr einfach miteinander synchronisiert. Der Zustand der Ursprungsressource wird an die übrigen Ressourcen weitergeleitet. Es fällt hier auf, dass der Agent in der Lage ist, zwischen den verschiedenen Zugriffen Zustände zu halten, die nicht in $\mathfrak{R}$ gespeichert worden sind.

Folgendes ist dabei zu beachten. Laut Teil 7.1.1 sollte ein Agent auf Zustände verzichten, damit Entscheidungen der Agenten stets gleich bleiben. Das Problem

eines abweichenden Agentenzustandes wird eliminiert, indem ein Agentenprogramm so gebaut ist, dass es unter der Bedingung, dass alle *Ressourcenzustände* (Z) gleich bleiben, es zum gleichen Agentenzustand kommt.

Im Fall der Mashups entscheidet der Zustand der Hauptressource, welche Aufrufe auf den Mashup-Ressourcen passieren. Ändert sich also der Zustand der Hauptressource nicht, dann werden exakt die gleichen Aufrufe durch den Agenten durchgeführt. Dies gilt natürlich nur unter der Voraussetzung, dass die Mashup-Ressourcen dann ebenfalls die gleiche Antwort auf gleiche Anfragen zurückgeben.

Mashups werden vor allem mit Methoden realisiert, die safe sind (lesend) und erfordern keine Konsistenz zwischen den Mashup-Ressourcen. Diese können unabhängig voneinander den Zustand ändern. Agenten haben auf die unter die Hauptressource untergeordneten Mashup-Ressourcen wahlfreien (ungeordneten) Zugriff. Es wird hier angenommen, dass es diese parallel adressiert werden können.

8.3 Verteilte Transaktionen

Während zwei verschiedene Mashup-Ressourcen auf Konsistenz verzichten, haben Transaktionen das Ziel diese zu bewahren. Die verteilten Transaktionen sind ein sehr altes Thema in der Informatik [EGL+76]. Sie werden immer wieder in verschiedenen Kontexten aufgegriffen. Es wurde auch auf der Basis von REST gezeigt, wie man Transaktionen konform zu diesem Architekturstil implementiert [MRMK09].

Eine Transaktion löst das Problem, Daten in einem verteilten System konsistent zu halten. Die Konsistenz bezieht sich auf eine übergreifende Aussage, die an verschiedenen unabhängigen Stellen im System ihre Gültigkeit bewahren muss. In einer ressourcenbasierten Architektur, die hier behandelt wird, bezieht sich die Transaktion auf die Konsistenz von Ressourcenzuständen, die über mehrere Ressourcen verteilt sind. Transaktionen sind allerdings innerhalb eines Architekturstils, der verteilt mit Daten operiert sehr umständlich zu benutzen. Sie erschweren die Programmierung von Agenten, indem eine zusätzliche Verwaltungsschicht hinzu kommt. Transaktionen führen auch dazu, dass sich Agenten, unabhängig von der Zugehörigkeit zur gleichen Agentengruppe, gegenseitig aussperren, um Konsistenz zu bewahren.

Im Gegensatz zu Mashups, halten sich Transaktionen anhand eines Protokolls, in dem festgelegt wird in welcher Reihenfolge die Ressourcen adressiert werden, an eine Ordnung, die für das Erreichen der Konsistenz eingehalten werden muss.

Es werden zuerst alle Verwaltungsmaßnahmen bezüglich Sperrungen durchgeführt, dann die eigentlichen Datenoperation, dann wird wiederum die Sperrung aufgeräumt. Dabei werden kooperative Prozesse auch nicht betrachtet und es wird nicht vom verteilten System ausgegangen, indem Agenten durch Ausfall ein Fehlverhalten für die gesamte Applikation provozieren könnten.

Bevor ein Ersatzmechanismus für die verteilte Transaktion eingeführt wird, müssen einige Eigenschaften für das Verhalten der Agenten erörtert werden. Dies wird in diesem Kapitel gemacht. Die Prinzipien, die hier vorgestellt werden, werden Grundlage für das nächste Kapitel 9 sein, wo in einigen Szenarien üblicherweise Transaktionen zum Einsatz kommen. Die dortigen transaktionsähnlichen Lösungen sind konform zum Paradigma der ressourcenbasierten Architekturen.

Eine solche transaktionsaktionsähnliche Lösung erfordert, dass ein Agent das Wechselspiel zwischen Agentenzuständen und Ressourcenzuständen geeignet behandelt, um die gesamte parallele Architektur nicht in Gefahr zu bringen, eine wesentliche Information zu verlieren und um redundante Ausführung in Agentengruppen zu unterstützen. Im Beispiel im Teil 7.1.1 wurde gezeigt, wie eine Abfolge von Aufrufen durch eine fehlerhafte Behandlung des internen Zustands dazu führt, dass ein anderer Agent aus der gleichen Gruppe nicht mehr funktionieren kann.

Es gibt auch Bedingungen, die für die Gestaltung des Agentenprogramms im besprochenen Architekturstil gelten. Diese werden nun Schritt für Schritt motiviert und näher untersucht.

8.4 Monotones Verhalten

Das Verhalten in einer verteilten Architektur wird mittels Kontrollmechanismen gesteuert und diese sind zu allererst auch Informationen, die genauso wie Daten und Datensätze persistent gehalten werden. Die Information, die zur Steuerung eines Systems beiträgt, muss jedoch strenger geregelt werden, als einfache Daten, die durch die Systeme der Architektur verarbeitet werden. Um Bearbeitungsreihenfolgen nachvollziehen zu können, darf man Information, die eine Eingabe für andere parallel stattfindende Berechnungen ist, nicht einfach entfernen. Dies würde dazu führen, dass Berechnungen zu unterschiedlichen Ergebnissen kommen.

Die Informationsausbreitung basiert auf dem Wechselspiel zwischen Kontrolle und Nachrichtenübertragung. In einer Architektur, die die *Monotonie* respektiert wird neue Information stets hinzugefügt. Auch ein Löschvorgang ist, auf diese Weise, eine neu hinzugefügte Information. Änderungen in Ressourcenzuständen

müssen also derart gestaltet werden, dass die Änderung nicht zu einem Vorgängerzustand führt, sonst würde das gleichbedeutend mit der Entfernung von Information sein, die nicht erlaubt ist.

Das einfache Entfernen einer Information aus einem System versetzt es in einen Zustand, indem die Information im System noch nicht vorhanden war. Das muss konsequenterweise ein Zustand gewesen sein, der *vor* dem Zustand existiert hat, der die Information enthält. In einer Architektur, die entkoppelt arbeitet, muss man annehmen, dass eine Information, die in einem beliebigen System gespeichert wurde, auch zu Entscheidungen beigetragen hat, die in dem gleichen System oder auch anderen Systemen zu bestimmten Entscheidungen geführt hat. Versetzt man ein System in den Zustand, wo eine Information fehlt, die zu diesen Entscheidungen geführt hat, bringt das potenziell Inkonsistenzen bezüglich der Zustände, die für Kontrolle im System verantwortlich sind.

Monotones Verhalten im System ermöglicht eine konsistente Sicht auf ein System, das Informationen lediglich akkumuliert, wie die Abbildung 8.2 verdeutlicht. Das verteilte System geht hierbei mit diskreten Schritten voran. Hat ein Agent seine Arbeit noch nicht verrichtet, dann ändert sich der verteilte Systemzustand nicht aus der Sicht der anderen Agenten. Erst wenn ein Arbeitsschritt abgeschlossen wird, persistiert der Agent die Änderungen in der Ressourcenwelt, sodass diese wahrgenommen werden können. Bis zu diesem Zeitpunkt verharrt das System im gleichen Zustand mit gleichen Voraussetzungen für redundant arbeitende Agenten.

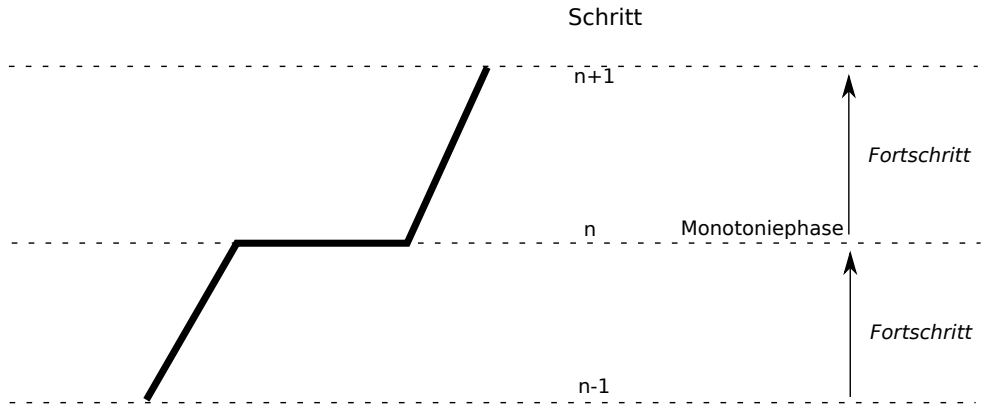

Abbildung 8.2: Illustration des monotonen Verhaltens in einer verteilten Architektur

Im Falle des Löschens von Informationen würde das bedeuten, dass Informationen als ungültig markiert werden, aber trotzdem erhalten bleiben, für Systeme, die noch (redundante oder parallele) Berechnungen auf alten Informationen ausführen, die, aus Konsistenzgründen, zum gleichen Ergebnis führen müssen.

Diese Eigenschaft im System erlaubt dem Entwickler, die Redundanz im System, die man mit Agentengruppen realisiert, völlig abstrakt zu behandeln. Im Endeffekt bedeutet das, dass der Entwickler sich darum nicht weiter kümmern muss. Weiterhin werden redundante Teillösungen, die später durch Agenten geliefert werden, nicht weiter berücksichtigt und haben keinen Einfluss auf die Gesamtberechnung. Diese werden entsprechend ignoriert, denn es kann bereits mit der ersten gelieferten Teillösung eine Fortführung der Berechnung erreicht werden, was in der Abbildung 8.3 dargestellt ist. Der nächste diskrete Schritt des verteilt arbeitenden Systems ist insgesamt früher erreicht worden. Die Gesamtberechnung läuft hiermit auch entlang der kürzesten und frühesten Ausführung, wobei die Einzelberechnungen voneinander entkoppelt sind und keiner weiteren überliegenden Koordination unterliegen.

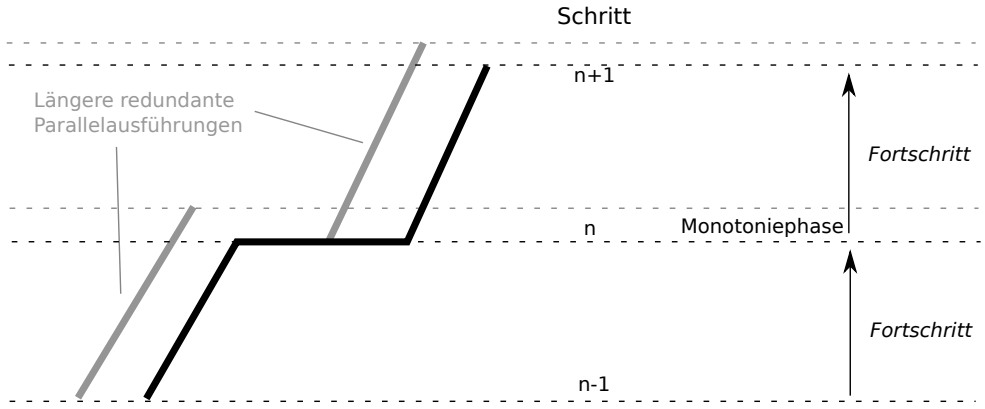

Abbildung 8.3: Monotones Verhalten und redundante Berechnungen

8.4.1 Monotonie bei der Instanziierung

Ähnlich wie das Löschen, führt eine falsche Reihenfolge von Operationen während der Instanziierung ebenfalls unter Umständen zur Inkonsistenz. Hierbei ist darauf zu achten, dass der nächste Zustand des Systems, in welchem ein neues Datum auftaucht atomar und gleichzeitig konsistent festgeschrieben werden muss.

Ein mögliches Verfahren, um dies zu realisieren, ist die Rückwärtsassoziation der Daten (siehe Beispielsequenz 8.1). Der Agent A_n instanziiert das Datum mit Hilfe der Ressource F zunächst so, dass es keinen Einfluss auf das System hat, indem er als einziger die *URI* u_{A_n} kennt. Das Hinzufügen in das System, um einen Vorwärtsschritt zu realisieren, ist die eigentliche indirekte Referenzierung des neuen Datum mit Hilfe einer Kontrollstruktur Q. Dabei muss das Hinzufügen der Referenz die monotone Eigenschaft haben. Ein Agent, der parallel die gleiche Arbeit

im gleichen Systemzustand macht, wird eine andere *URI* erzeugen, deren identische Bedeutung auf einem anderen Wege festgestellt werden muss. Im Falle der Warteschlangentypen würde das dazu führen, dass anstatt des Eingangskriteriums T, die allgemeinere Variante V gewählt worden wäre, die die Identität anhand von d feststellt, anstatt der *URI*, die für jeden redundanten Agenten verschieden sein würde.

$$\begin{aligned} &A_n : \text{create}_F(\{d\}) : \{u_{A_n}\}, \\ &A_n : \text{queue}_Q(\{u_{A_n}, d\}) \end{aligned} \tag{8.1}$$

8.5 Monotononie der Informationsausbreitung

Im Teil 8.4.1 konnte man sehen, dass die Monotonie alleine mit Hilfe der Spezifikation des Agentenverhaltens nicht eingehalten werden kann. Dazu gehört die Wechselwirkung aus dem geordneten Verhalten und der synchronisierenden Datenstruktur.

Im Teil 5.4.3 wurde die Eigenschaft der Idempotenz ausführlich beleuchtet und dann später zu der Eigenschaft der Monotonie (im Teil 5.4.5) erweitert. Die Monotonie modelliert die „natürliche“ Form der Ausbreitung von Information in einer ressourcenbasierten Architektur. Die Abbildung 8.4 zeigt das Prinzip, wie sich die Information an Beispiel von Web ausbreitet.

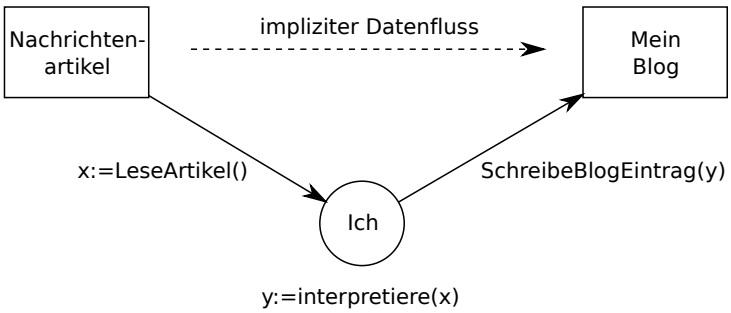

Abbildung 8.4: Beispiel der Ausbreitung von Information im Web.

In der abstrakten Form lässt sich diese Ausbreitung auf die Weise illustrieren, die in der Abbildung 8.5 gezeigt wird.

Man ist sich deswegen vielleicht nicht ganz bewusst, wenn man das World Wide Web betrachtet, denn die Kontrolle der dortigen Vorgänge ist nicht ausdrücklich in Ressourcen gespeichert. Im Beispiel des WWW betrachten wir die Agenten meistens als Menschen, die einen Web-Browser bedienen und da Menschen

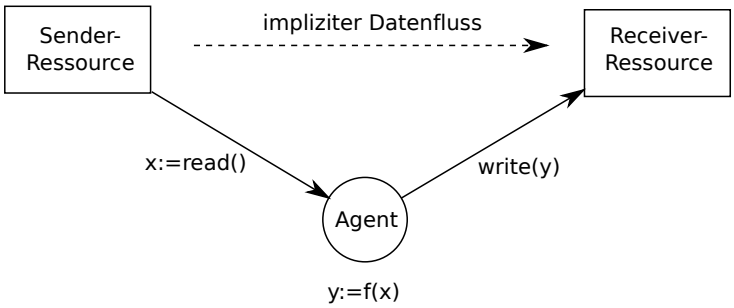

Abbildung 8.5: Abstrakte Form des Informationsausbreitung.

in ihrer Erinnerung sehr viele Zustände speichern können, sind sie im Grunde vergleichbar mit Agenten, die über mehrere Jahre laufen (eigentlich lebenslange Laufzeit). Ein solcher langlebiger Agent speichert in seinen internen Zuständen welche Information er bereits konsumiert hat und beurteilt anhand dieser Zustände, welche Information er noch nicht kennt. Der Zustand, der die Funktionsweise der Ausbreitung von Information rekonstruiert, existiert „im Kopf“ des Agenten, wie die Abbildung 8.6 verdeutlicht. Der ausgefüllte Kreis in der Abbildung markiert den Speicherort für die Kontrolle der Informationsausbreitung.

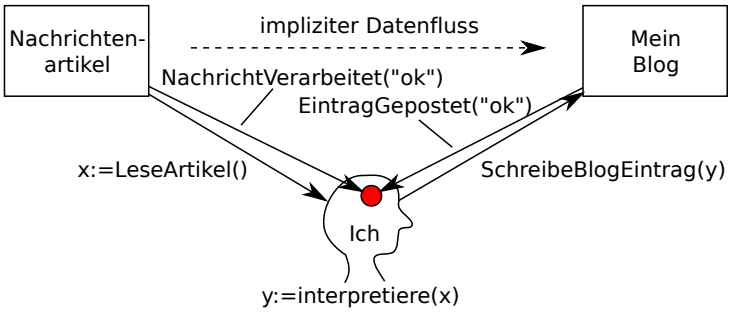

Abbildung 8.6: Kontrolle der Informationsausbreitung anhand eines menschlichen Agenten.

Dass die Information an einer Stelle weiter gereicht worden ist, sollte in Architekturen, die Kooperation erlauben, feststellbar für andere Agenten gestaltet werden. Und die einzige Art uns Weise, wie man das realisieren kann, ist die Kontrolle in $\mathfrak{R}$ zu verwalten. In einfachen Fällen kann das wie in der Abbildung 8.7 gestaltet werden, dass die Senderressource oder die Empfängerressource die Kontrolle mitführt und damit den Agenten von der Aufgabe der Speicherung der Kontrollzustände befreit.

Außerdem ist es natürlich möglich, die Kontrolle in eine separate Ressource auszulagern, die explizit dafür gedacht worden ist. Dabei muss die Semantik allen Agenten klar sein und sie müssen sich an bestimmte Reihenfolgen halten, wenn sie mit den kontrollierten Daten umgehen.

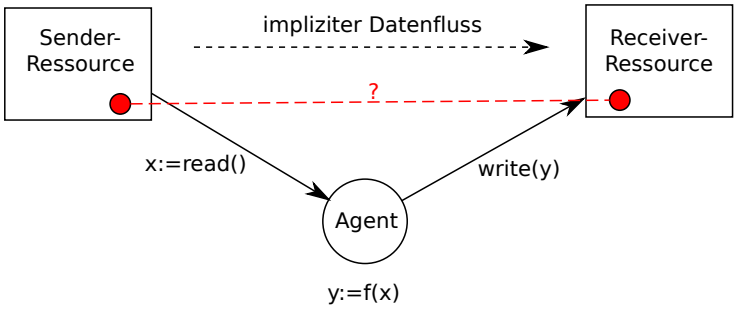

Abbildung 8.7: Kontrolle der Informationsausbreitung im Falle von Kooperation.

Diese Reihenfolgen, die als Protokoll verstanden werden können, werden nun genauer untersucht.

8.6 Sequenzen in Agentengruppen

In diesem Teil wird die Idempotenzeigenschaft einzelner Methodenaufrufe untersucht. Ferner werden Aussagen über Aneinanderreihungen idempotenter Methodenaufrufe gemacht und Schlüsse gezogen, wie man zu einer konsistenten Informationsübertragung gelangen kann. In diesem Abschnitt wird das Verhalten von redundanten Agenten einer einzelnen Agentengruppe betrachtet.

Sequenzen, die Monotonieeigenschaften aufwiesen wurden bereits stillschweigend im letzten Kapitel im Teil 7.2.1 eingeführt. Ohne weiter motiviert zu werden, wurde dort die Kontrolle in einer „acknowledge“-Ressource durchgeführt. Das Prinzip, wie mit der Kontrolle dort umgegangen wurde, lässt sich generalisieren. Und dies wird in den folgenden Abschnitten genauer erörtert.

8.6.1 Disjunkte Ein- und Ausgaberessourcen

Die Sequenz (8.2) wird von einem Agenten ausgeführt, der Ressource mit der Referenz u für Eingabe gebraucht und mittels der Funktion f Ausgaben berechnet, die in Ausgaberessourcen mit den Referenzen v und w gespeichert werden.

$$\begin{aligned} &\text{read}_u() : \{x\}, \\ &\text{write}_v(f(x)), \\ &\text{write}_w(f(x)) \end{aligned} \tag{8.2}$$

Offensichtlich kann eine ganze Agentengruppe diese Berechnung konsistent durchführen, weil die Ein- und Ausgaben hier getrennt sind und es lokal zum gleichen Schluss kommt und zu gleichen Ergebnissen in v und w. Voraussetzung ist natürlich der gleiche Zustand Z für die Methode $\text{read}_u \in M_{u,Z}$.

8.6.2 Beschreiben der Eingangsressource

Die Separation der Ein- und Ausgabemenge trifft nicht immer zu, deswegen ist es interessant zu schauen, wie die Konsistenz beeinflusst wird, wenn die Eingaberessource beschrieben wird. Dazu gibt es die folgenden Möglichkeiten: Ressource mit Referenz u wird überschrieben (siehe (8.3)) oder u wird invalidiert („gelöscht“; siehe (8.4)).

$$\begin{aligned} &\text{read}_u() : \{x\}, \\ &\text{write}_u(f(x)) \end{aligned} \tag{8.3}$$

$$\begin{aligned} &\text{read}_u() : \{x\}, \\ &\text{delete}_u() \end{aligned} \tag{8.4}$$

Obwohl die Abläufe in einem verteilten System relativ schlecht zu verstehen sind, würde eine Agentengruppe die beiden Aufrufe zu einem Zeitpunkt konsistent durchgeführt haben, sodass sie die Gesamtwirkung final erzielen. Natürlich würde ein Agent, der nach den beiden Aufrufen zum Einsatz kommt ein anderes Verhalten im System verursachen, aber Fakt ist, dass die Arbeit des vorigen Agenten (mindestens ein Mal) konsistent abgeschlossen worden ist, um so weit zu kommen.

Alternativ kann man ein etwas konservativeres Verhalten der Agenten programmieren, welches mit Indirektion arbeitet, um das Abarbeiten eines Datums nachvollziehen zu können. Das könnte man auf die Weise realisieren, die im Beispiel (8.5) gezeigt ist.

$$\begin{aligned} &\text{read}_p() : \{u\}, \\ &\text{read}_u() : \{x\}, \\ &\text{write}_v(f(x)), \\ &\text{write}_p(\{v\}) \end{aligned} \tag{8.5}$$

In diesem Beispiel ist die Ressource p, die Indirektion benutzt, sowohl auf der Ein- wie auch auf der Ausgabeseite im Agentenprogramm. Das Datum selbst wird von Ressource u nach v übertragen, nachdem die agenteninterne Funktion f angewendet worden ist. Eine Ressource hinter der *URI* p verwaltet die Informationsübertragung und steuert den Agenten anhand einer dort abgespeicherten *URI*.

8.6.3 Verletzung der Monotonieeigenschaft in Agentengruppen

Vielleicht wird dem aufmerksamen Leser klar, dass sich hier eine bestimmte Regel bei der Übertragung herauskristalisiert. Die Ein- und Ausgaberessourcen wurden schon von Anfang an aus einem bestimmten Grund getrennt betrachtet.

Man kann ganz leicht eine Sequenz idempotenter Methoden bilden, die keine Monotonie einhält. Man sehe sich das Beispiel (8.6) an.

$$\begin{aligned} &\text{read}_u() : \{x\}, \\ &\text{write}_u(f(x)), \\ &\text{write}_v(f(x)) \end{aligned} \tag{8.6}$$

Hier wird die Eingaberessource nicht im letzten Schritt beschrieben, sodass für zwei Agenten aus der gleichen Agentengruppe zwei alternative Berechnungen möglich werden. Die erste führt am Ende $\text{write}_v(f(x))$ aus. Die zweite führt $\text{write}_v(f(f(x)))$ aus. Es ergibt sich eine Inkonsistenz, weil die erste Berechnung eventuell niemals festgeschrieben wird, wenn der betreffende Agent im ungünstigen Fall ausfällt (nach dem zweiten Aufruf).

Die plausible Lösung für das Szenario, ist die Ressource, die für die Eingabe nötig ist und bei der Ausgabe beschrieben wird, ans Ende der Sequenz zu stellen. Und tatsächlich ist dieser Fall eliminiert, wie das Beispiel (8.7) zeigt. Die Kontrolle der Informationsübertragung wird hier auf die Ressource hinter der *URI* u verlagert.

$$\begin{aligned} &\text{read}_u() : \{x\}, \\ &\text{write}_v(f(x)), \\ &\text{write}_u(f(x)) \end{aligned} \tag{8.7}$$

8.7 Sequenzen agentengruppenübergreifend

Im Abschnitt 8.6 wurde das Agentenverhalten auf eine einzelne Gruppe eingeschränkt. Bei Zusammenstellung eines oder mehrerer Systeme, ist es aber der Fall, dass Information stets weiter gereicht wird und von einer Agentengruppe an eine andere konsistent zur Verfügung gestellt werden muss.

8.7.1 Disjunkte Ein- und Ausgaberessourcen

Die erstaunliche Einsicht ist, dass bei der Beteiligung mehrerer Agentengruppen, es nicht so einfach ist, die Konsistenz zu bewahren.

Unter der Annahme, dass von einer Agentengruppe die sich als harmlos darstellende Sequenz 8.8 durchgeführt wird (es ist das gleiche Beispiel wie (8.2) in 8.6.1).

$$\begin{aligned} &\text{read}_u() : \{x\}, \\ &\text{write}_v(f_1(x)), \\ &\text{write}_w(f_1'(x)) \end{aligned} \tag{8.8}$$

Übernimmt eine zweite Agentengruppe die Ausgaben der ersten Agentengruppe als Eingaben und führt die folgende Sequenz in 8.9 aus, dann ist die Konsistenz der Berechung nicht gewährleistet.

$$\begin{aligned} &\text{read}_v() : \{y\}, \\ &\text{read}_w() : \{z\}, \\ &\text{write}_t(f_2(y, z)) \end{aligned} \tag{8.9}$$

Eine solche Ausführungssequenz wäre (8.10).

$$\begin{aligned}
&\text{read}_u() : \{x\};\\
&\text{write}_v(f_1(x));\\
&\text{read}_v() : \{y\};\\
&\text{read}_w() : \{z\};\\
&\text{write}_w(f_1'(x));\\
&\text{write}_t(f_2(y,z))
\end{aligned} \tag{8.10}$$

Die Sequenz erzeugt ein anderes Ergebnis in Ressource unter *URI* t, als wenn die Seqenzen (8.8) und (8.9) hintereinander ausgeführt worden wären.

8.7.2 Indirektion für atomare Schreibvorgänge

Das Problem im Abschnitt zuvor liegt darin, dass die Schreibvorgänge nicht atomar gestaltet sind. Dieses Fehlverhalten kann man mittels Indirektion und Synchronisierung der beiden Ausgaben reparieren.

Dazu verlagert man die Kontrolle auf eine Ressource mit Referenz p, die beide Werte als *URI*s speichert, wie das die Programme in (8.11) und (8.12) für die beiden Agentengruppen zeigen. Zudem werden die beiden Werte in der neuen Kontrollressource logisch zusammengefasst.

$$\begin{aligned}
&\text{read}_u() : \{x\},\\
&\text{write}_v(f(x)),\\
&\text{write}_w(f(x)),\\
&\text{write}_p(\{v,w\})
\end{aligned} \tag{8.11}$$

$$\begin{aligned}
&\text{read}_p() : \{v,w\},\\
&\text{read}_v() : \{y\},\\
&\text{read}_w() : \{z\},\\
&\text{write}_t(f(y,z))
\end{aligned} \tag{8.12}$$

Diese beiden Sequenzen sind immer noch anfällig für „Lost Updates“, die im Teil 7.2 angesprochen worden sind. Man erkennt an dieser Stelle bereits, dass eine monotone Sequenz, die nicht anfällig für „Lost Updates“ ist, erst mit Hilfe von datenstrukturgetriebenen Ressourcen zum konsistenten Verhalten führen kann. Die Zusammenwirkung des hier besprochenen Verhaltens und der datenbasierten Ressourcen wird für einige interessante Beispiele im Kapitel 9 besprochen.

8.8 Definition monotoner Sequenzen

Eine *monotone Sequenz* ist eine Sequenz *idempotenter* und *monotoner Methoden*, die innerhalb einer Agentengruppe alle Verzahnungen und alle Ausfälle von Agenten toleriert und entweder die beschriebene und beabsichtigte Wirkung beim erfolgreichen Abschluss der Sequenz erreicht oder die Sequenz der Methodenaufrufe wurde noch nicht abgeschlossen. Schlussendlich, unter der Annahme, dass die Agentgruppe stets irgendwann aktiv wird, wird auch die monotone Sequenz irgendwann abgeschlossen.

Es wird zunächst gezeigt, warum *Idempotenz* als Kriterium erforderlich ist, bevor die Konsistenz behandelt wird, wo auch *Monotonie* der Methoden erforderlich ist.

8.8.1 Ausschluss von nichtidempotenten Methoden

Eine monotone Sequenz kann keine *nichtidempotenten Methoden* enthalten. Ohne weitere Einschränkung ist klar, dass lesende Methode „safe“ sind und damit idempotent. Es geht also um die Eigenschaft der zustandsändernden Methoden (auf Ausgaberessourcen).

Sei $m \in M_{u,Z}$ eine Methode mit $U(R) = \{u\}$, die beim Aufruf $m(p)/[Z \to Z']$ nicht idempotent ist. Dann existiert ebenfalls die *nicht idempotente* Methode $m \in M_{u,Z'}$, die beim Aufruf $m(p)/[Z' \to Z'']$ in einen Zustand Z'' gelangt. Diese Verzahnung von Aufrufen für zwei Agenten A_1 und A_2, kann folgendermaßen aussehen.

$$\ldots; \tag{8.13}$$
$$A_1 : m(p)/[Z \to Z'] : y';$$
$$A_2 : m(p)/[Z' \to Z''] : y'';$$
$$\ldots \tag{8.14}$$
$$\tag{8.15}$$

Diese Verzahnung ist offensichtlich eine andere, wenn A_1 die Methode $m(p)$ ausschließlich allein benutzt. A_1 allein bringt die Ressource R in Zustand Z'. In Kooperation mit A_2 jedoch in den Zustand Z'', was laut Definition einer monotonen Sequenz nicht erlaubt ist.□

8.8.2 Abschlussregel für monotone Sequenzen

Seien $r_1(p_1)/[Z_1 \to Z_1] : y_1, \ldots, r_n(p_n)/[Z_n \to Z_n] : y_n$ lesende Methodenaufrufe in einer monotonen Sequenz, die Eingaben für die Berechnung in einer Gruppe

von Agenten liefern. Sei ferner ein $w_i(p_i)/[Z_i \rightarrow Z_i']$ ein Aufruf, welcher einen Zustandsübergang in diesen Eingaben verursacht und zu $r_i(p_i)/[Z_i' \rightarrow Z_i'] : y_i'$ mit $1 \leq i \leq n$ und $y_i \neq y_i'$ führt. Der Methodenaufruf $w_i(p_i)$ darf ausschließlich der letzte (nichtredundante[1]) Aufruf einer monotonen Sequenz sein. Es wird weiterhin angenommen, dass die Eingaben $y_1, \ldots, y_n$ essentiell für die agenteninterne Berechnung sind. Das bedeutet, dass sie sich auf die Ausgabe auswirken.

Sei $w_i(p_i)/[Z_i \rightarrow Z_i']$ nicht der letzte schreibende Methodenaufruf, dann ist die Sequenz noch nicht vollständig abgearbeitet worden und es gibt noch mindestens ein $w_j(p_j)$ (dabei kann $j > n$ gelten, also muss nicht unbedingt die lesenden Methodenaufrufe beeinflussen), welches durch einen Agenten noch aufgerufen werden muss, wenn er diese Sequenz ausführt. Nimmt man an, dass der Agent A_1 die Sequenz so durchführt, dass alle Eingaberessourcen die Antwort $y_1, \ldots, y_n$ liefern und der Agent A_2 bekommt anstatt y_i ein y_i' als essentielle Eingabe zur Verarbeitung der Daten. Führt A_2 den Aufruf $w_i(p_i')/[Z_i' \rightarrow Z_i'']$ unter Umständen mit einem $p_i' \neq p_i$ aus bevor A_1 die Sequenz abschließt, dann weicht die Wirkung der Gesamtsequenz der Agentengruppe ab, wegen $Z_i' \neq Z_i''$, wenn A_1 die Sequenz abschließt, was die Definition der monotonen Sequenz nicht zulässt.□

Zur Illustration des Sachverhalts dient die Abbildung 8.8. Liefert eine Ressource eine Eingabe, die während der gesamten Sequenz verändert wird, dann darf die Änderung nicht vor dem Ende der Sequenz geschehen.

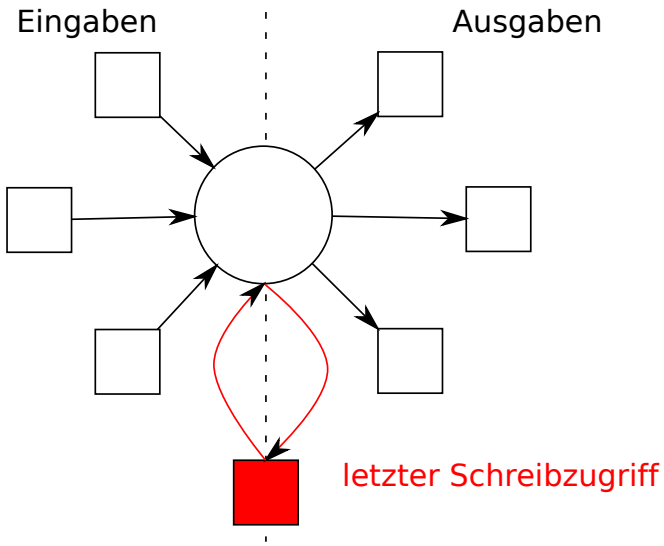

Abbildung 8.8: Abschlussregel bei der Bildung von monotonen Sequenzen

[1] w_i könnte, weil es idempotent ist, mehrmals am Ende der Sequenz auftauchen, dies ist jedoch eine redundante Operation

8.8.3 Relaxation der Reihenfolge für lesende Methoden

Ohne weitere Einschränkung darf der Agent alle lesenden Methodenaufrufe vor die schreibenden Aufrufe stellen. Trivialerweise gilt für ein Agentenprogramm, dass ein zustandsändernder Methodenaufruf $w_j(q_j)$ in der Sequenz nur dann ausgeführt werden kann, wenn alle essentiellen Eingaben gelesen worden sind, die für den Aufruf nötig sind. Stellt man alle lesenden Zugriffe $r_i(p_i)$ vor die zustandsändernden, sieht das folgendermaßen aus (8.16).

$$r_1(p_1) : y_1, \ldots, r_n(p_n) : y_n, w_{k_1}(q_1), \ldots, w_{k_m}(q_m) \tag{8.16}$$

Da diese Sequenz ebenfalls alle für die Berechnung der $w_{k_j}(q_j)$-Aufrufe essentiellen Eingaben $y_1, \ldots, y_n$ enthält, ist das Gesamtverhalten der Sequenz gleich im Falle eines einzelnen Agenten. Laut Teil 8.8.2 darf höchstens $w_{k_m}(q_m)$ die Eingaben ändern.

8.8.4 Konsistenz der monotonen Sequenz

Es wurde noch nicht gezeigt, wann eine Sequenz mit mehreren Agenten einer Agentengruppe in der verteilten Architektur eine monotone Sequenz ist und die Konsistenz bewahrt. Dazu reicht die Bedingung der Idempotenz der aufzurufenden Methoden allerdings nicht aus, wie sich gleich herausstellt.

Sei in (8.16) $A_f \in \mathfrak{A}\ A_f : w_m(q_m)$ der Aufruf, der die Antwort y_i der Eingaberessource $r_i(p_i)$ durch seine Zustandsänderung zu y_i' verändert. Es ist klar, dass bevor $w_m(q_m)$ ausgeführt wird, zwei Agenten A_1 und A_2 wegen ihrer Zustandslosigkeit und gleichen Eingaben das gleiche schließen werden. Interessant ist vielmehr der Fall, in dem Agent A_f den Aufruf $w_m(q_m)/[Z_m \rightarrow Z_m']$ ausgeführt hat und das Verhalten der Agenten A_1 und A_2, wobei A_1 die Änderung der Eingabe $r_i(p_i)/[Z_m \rightarrow Z_m] : y_i$ nicht erfährt und $A_2\ r_i(p_i)[Z_m' \rightarrow Z_m'] : y_i'$ nach der Änderung durchführt.

Mit Hilfe der Relaxation in 8.8.3 darf man einfachheitshalber annehmen, dass weder Agent A_1 noch A_2 eine Zustandsänderung verursacht haben, wenn sie noch in der Lesephase sind. Der Agent A_1 wird nach dem Abschluss aller $r_i(p_i)$ genau die gleiche Gesamtwirkung der monotonen Sequenz verursachen, wie das der Agent A_f gemacht hat. Es werden konsequenterweise die gleichen zustandsändernden Methodenaufrufe $w_j(q_j)$ auf den Zielressourcen ausgeführt.

An dieser Stelle sollte klar werden, dass Idempotenz allein hier nicht ausreichend ist, um Konsistenz beim Schreiben zu bewahren. Der Agent A_2, der bereits auf den neuen Eingaben arbeitet, die A_f geschrieben hat, wird in den Ausgaben für

Race Conditions mit dem Agent A_1 sorgen (siehe: letzte beiden Zeilen von 8.17). Idempotenz garantiert zwar in konsekutiver Ausführung für das gleiche Resultat, aber an dieser Stelle wird tatsächlich die *Monotonie* der Methodenaufrufe verlangt, die den Reihenfolgenkonflikt der letzten beiden Aufrufe in diesem Beispiel behandelt.

$$
\begin{aligned}
&\ldots \\
&A_f : w_j(q_j) \\
&\ldots \\
&A_1 : r_i(p_i)/[Z \to Z] \\
&A_f : w_m(q_m)/[Z \to Z'] \\
&A_2 : r_i(p_i)/[Z' \to Z'] \\
&\ldots \\
&A_2 : w_j(q'_j) \\
&A_1 : w_j(q_j)
\end{aligned}
\tag{8.17}
$$

Mit Hinzunahme der Monotoniebedingung (siehe (5.10) in 5.4.5) von $w_j(q_j)$, ergibt sich zusätzlich die Eigenschaft, dass ein verspäteter redundanter Aufruf keine Zustandsänderung in der Ressource verursacht. Effektiv erhalten A_1 und A_2 bezüglich der Aktualität t eine Ordnung, die von der Ausgaberessource verwaltet wird. Auf diese Weise gibt es keine Race Condition während der schreibenden Aufrufe, die A_1 und A_2 durchführen. Der verspätete Methodenaufruf $A_1 : w_j(q_j)$ wird im Falle der Monotonie keine Zustandsänderung herbeiführen, weil exakt der gleiche Aufruf bereits von A_f getätigt wurde.

Monotone Eigenschaften für Methoden wurden bereits im Teil über Warteschlangen besprochen (siehe 7.7). Die Anwendungen der Erkenntnisse werden in den folgenden Kapiteln 9 und 10 illustriert. An dieser Stelle sollte jedoch klar werden, dass für die Konsistenz der Informationsübertragung die *monotonen Sequenzen* nötig sind, aber erst durch den Einsatz der Ressourcen, die monotone Methoden anbieten, wird das Konzept hinreichend vervollständigt.

8.9 Komposition monotoner Sequenzen

Die Abschlussregel für monotone Sequenzen (siehe Teil 8.8.2) verlangt, dass Zustandsmodifikationen in den Eingaberessourcen ausschließlich im letzten Schritt gemacht werden dürfen. Interessanterweise lässt sich mit einem Hilfskonstrukt eine Komposition erreichen, die diese Regel zwar nicht widerlegt, aber eine Möglichkeit anbietet, die monotone Sequenz logisch zu verlängern.

Die Idee dahinter ist, eine Sequenz, nach dem abschließenden Beschreiben, mit genügend Entkopplung auszustatten, sodass die monotone Sequenz von zwei Agentengruppen ausgeführt werden kann. Es ist hierbei trivial zu zeigen, dass das Verhalten von zwei Agentengruppen mittels einer Agentengruppe simuliert werden kann, indem die eine Gruppe die Programme für beide Arten von Verhalten wahlweise ausführt.

Schaltet man zwischen die beiden monotonen Sequenzen eine Warteschlange Q als Entkopplungsmechanismus, ist es möglich, die Sequenz, unter der Bewahrung der Konsistenz des Gesamtverhaltens, in der Agentengruppe zu verlängern.

$$\frac{\begin{array}{l}\ldots,\\ \texttt{queue}_Q(\{d, Id\}),\end{array}}{\begin{array}{l}\texttt{peek}_Q() : \{d', Id'\},\\ \ldots,\\ \texttt{unqueue}_Q(\{Id'\})\end{array}} \tag{8.18}$$

In (8.18) ist ein Beispiel einer monotonen Sequenz, die auf den ersten Blick die Bedingung der Abschlussregel aus Teil 8.8.2 verletzt, weil nach der Methode queue_Q die Ressource Q lesend benutzt wird mit dem Methodenaufruf peek_Q. Da queue_Q nicht die letzte Methode ist, widerspricht das scheinbar der Abschlussregel.

Zur Verdeutlichung, warum die Abschlussregel ihre Gültigkeit behält, wurde in die Sequenz optisch eine Trennung zwischen den beiden einzelnen monotonen Sequenzen eingezeichnet. Durch die Entkopplung mit Hilfe der Warteschlange Q, wird im ersten Teil eine Ausgabe gemacht, die die Eingaben der zweiten monotonen Sequenz nicht (direkt) beeinflusst.

Bei der Konstruktion dieser Entkopplung ist zuzusichern, dass die längere Sequenz in zwei eigenständige Sequenzen zerfallen kann und zwar so, dass an der so geschaffenen Auftrennung ein neuer Agent übernehmen kann. Das kann der Agent natürlich nur dann tun, wenn jeglicher Zustand aus dem ersten Teil der Sequenz entweder persistiert wird oder verworfen werden kann.

Und dies ist genau der Grund warum es funktioniert. Die beiden Sequenzen zerfallen so, dass Agenten über die Trennung hinaus keinen internen Zustand halten müssen, um die Arbeit fortführen zu können.

8.10 Diskussion der Ergebnisse

In diesem Teil wurden *monotone Sequenzen* vorgestellt, die ein lokales Verhalten für Agentengruppen formulieren, unter der Bedingung, dass Konsistenz bei der Übertragung von Information bewahrt wird.

Mit Hilfe der *Abschlussregel* aus Teil 8.8.2, ist es möglich eine atomare Änderung im System zu erzeugen, die völlig ohne Mechanismen zur Sperrung auskommt. Diese Regel stellt eine Art Bauplan für ein Protokoll zur Informationsübertragung, der eingehalten werden muss, um zwei Systeme (oder zwei Agentengruppen) kooperativ miteinander zu komponieren.

Im Teil 8.8.4 wurde gezeigt, warum datenbasierte Kontrolle anhand von *generischen Ressourcen*, die lediglich *Idempotenz* liefern können, nicht ausreicht. Um Systeme komponieren zu können, bedarf es der Benutzung von Ressourcen, die monotone Eigenschaften aufweisen, wie die im Teil 7.7 eingeführten Warteschlangentypen.

Kapitel 9

Simulation

In diesem Kapitel wird gezeigt, dass eine Architektur, die nach den Prinzipien funktioniert, die in den vorhergehenden Kapiteln motiviert worden ist, in der Lage ist, einige mächtigere Systeme zu simulieren. Allerdings, soll das nicht so verstanden werden, dass die eingeführten Konzepte ein Workflow-Netz oder Petrinetze ersetzen soll. Es ist lediglich ein Hiweis, dass eine ressourcenbasierte Architektur mächtig und damit generell genug ist, die dort zu Grunde liegenden Konzepte zu unterstützen.

Die stark synchronisierte und gekoppelte Arbeitsweise der Workflow- und Petri-Netze ist ein vollkommen anderer Ansatz und hat durchaus positive Eigenschaften für Prozesse, die auf diese Arbeitsweise oft angewiesen sind.

9.1 Workflow-Netz

Ein Workflow-Netz besteht in der Modellansicht aus einer größeren Vielfalt von Modellelementen, die den Kontrollfluss eines Prozesses steuern können. In diesem Teil werden die einzelnen Elemente zur Prozesskontrolle durch die Verfahren aus in dieser Arbeit eingeführten Architektur und ihren Prinzipien simuliert.

Neben kooperativen Prozessen sind übliche Geschäftsprozesse seitens ihres Verhaltens etwas anders gestaltet. Sie unterliegen dem Prinzip des Synchronisierens in fast jedem Schritt. Die hier besprochene Simulation scheint deswegen etwas aufwändig, weil die Umkehr der Paradigmen dazu zwangsweise führt. In der Praxis bedeutet das, dass man sich gut überlegen sollte, ob ein Prozess entkoppelt und frei verteilt ist, oder von der Synchronisierung dominiert wird und sehr strikt sequenzenorientiert organisiert wird.

Ein weiteres Merkmal, welches Workflow-Netze gut unterstützen, ist die Wiederverwendbarkeit von Aktivitäten, welche hier, in der ressourcenbasierten Architektur, äquivalent zu einem Untersystem ist. Um zu unterscheiden, welche

Daten aus welchen verteilten externen Ressourcen kommen und wohin sie aus dem System gespeichert werden sollen, müssen diese mit Hilfe eines Mechanismus, welcher Instanzen nachverfolgt ausgestattet werden. Dies impliziert auch, dass jedes Informationselement, das hier weiter gereicht wird eine *Identifizierung der Prozessinstanz* mitführen muss.

Um die Simulation von Workflow-Netzen strukturiert untersuchen zu können, wird als Vorlage das bekannte Werk „Workflow Patterns“ [vdAtHKB03] genommen. Die wichtigsten Muster, die sich auf die Kontrolle beziehen („control patterns“), werden vorgestellt und entsprechend der bestehenden Mittel des ressourcenbasierten Architekturstils simuliert.

Die Simulation der einzelnen Muster läuft grundlegend so ab, dass Agenten Workflow-Aktivitäten durchführen und die Ressourcen in $\mathfrak{R}$ entsprechend eine Zustandsänderung erfahren. Es wird hier angenommen, dass Daten und Kontrolle getrennt sind und es werden in der Simulation nur die Ressourcen betrachtet, die zur Steuerung des Prozesses beitragen (Flusskontrolle). Die tatsächlich ausgeführten Aktivitäten tauchen hier deswegen nicht auf und konsequenterweise sind die eigentlichen Datenressourcen ebenfalls nicht aufgeführt. Es reicht nachzuweisen, dass der Aspekt der Kontrolle in die Ordnung gebracht werden kann, die die Kontrollanweisungen der Workflow-Sprachen vorschreiben.

9.1.1 Sequenz

Wie schon am Anfang im Teil 3.2 angekündigt, ist die einfachste Form der Prozesssteuerung in Workflows die Sequenz. Deswegen hat sie kein explizites grafisches Element in Workflow-Netzen, sondern wird allgemein mit einem Verbindungspfeil zwischen zwei Aktivitäten symbolisiert.

Die entsprechende sequenzielle Ankopplung eines Systems gestaltet sich mit einer *IST/ON*-Warteschlange sehr einfach (siehe Abbildung 9.1). Hat ein Agent $A_1 \in G_{\text{in}}$ eine Aufgabe erledigt, kann er die Fertigstellung der Aufgabe in der Warteschlange mitteilen. Ergebnisse anderer Agenten aus G_{in} werden mit Hilfe von Monotonie unterdrückt. Auf der Gegenseite wird die Agentengruppe G_{out} aktiviert und kann die Weiterbearbeitung übernehmen.

Das Kriterium T kann auf der Eingangsseite der Warteschlange natürlich nur dann zur Entscheidung genommen werden, wenn Ergebnisse aus gleichen Agentinstanzen vollkommen identisch sind. Sollte ein Teil des Datums abweichen können (das kann diverse Gründe haben), muss man den Warteschlangentyp *ISV/ON* benutzen und das Kriterium V entsprechend der passenden Ordnung auswerten.

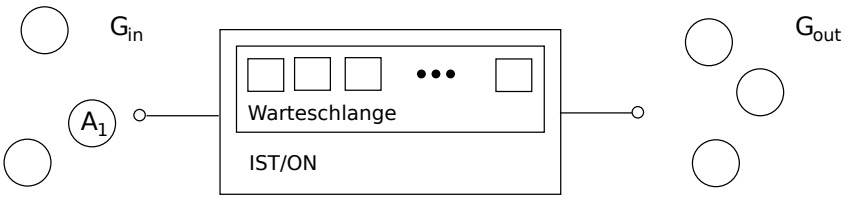

Abbildung 9.1: Warteschlangentyp IST/ON *zur Simulation von Sequenzen*

9.1.2 Parallele Verzweigung

Die parallele Verzweigung braucht zur Simulation keine bestimmte Datenstruktur zur Unterstützung. Ist ein Ergebnis in einer Ressource zur Verfügung gestellt worden können beliebig viele Agenten eine angekoppelte Berechnung starten. Das Vorgehen dazu demonstriert die Abbildung 9.2. Es ist ein ganz natürliches Prinzip in den Architekturen, die hier behandelt werden. In BPEL-Terminologie entspricht die parallele Verzweigung dem `<flow>`-Element und wird explizit angegeben.

Propagiert man ein Datum in n verschiedene Zweige mit Agentengruppen $G_1, \ldots, G_n$ in separate Ressourcen, werden die einzelnen Zweige selbstständig und entkoppelt arbeiten. Um an dieser Stelle das Verständnis etwas zu erleichtern, wurden die Agentengruppen zugehörig zu dem jeweiligen Pfad der Informationsausbreitung abgebildet. Die Information wird hier von R_in auf die parallelen Zweige $R_{\text{out},1}, \ldots R_{\text{out},n}$ verteilt, um sie von dort an entkoppelt bearbeiten zu lassen.

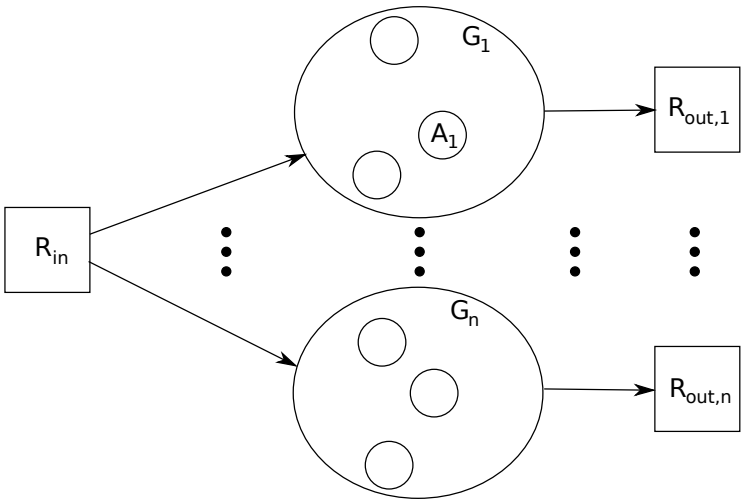

Abbildung 9.2: Simulation einer parallelen Verzweigung

Diese Art von Informationstransport hat eine starke Ähnlichkeit zu dem Newsfeed-Aggregator aus Teil 3.13 mit einer Quellressource. Und in der Tat ist die Funktionsweise eines solchen 1-Newsfeed-Aggregators semantisch gleich. Aus einem Datum werden n Subsysteme angesteuert. Sie arbeiten nach der Verzweigung alle unabhängig voneinander. Der eigentliche Unterschied ergibt sich erst, wenn man die anschließende Synchronisierung, die im nächsten Teil besprochen wird, berücksichtigen möchte.

9.1.3 Synchronisierung mit *AND*-Joins.

Um eine *AND*-Join-Synchronisierung durchzuführen, muss man zuvor, bei einer parallelen Verzweigung (siehe 9.1.2), die Zahl der ausgehenden Zweige kennen, die bei BPEL statisch vorgegeben ist. Hier kann man nicht einfachheithalber die Zahl der bereits verzweigten Prozesse anhand von dynamisch regelbarer Anzahl von Abzweigungen feststellen. Denn ein Agent der einen Zweig mit einem neuen $R_{\text{out,j}}$ anstarten möchte, kann erst sehr spät erscheinen und den Zweig anstarten, sodass es nicht feststellbar ist, wann ein *AND*-Join synchronisieren soll.

Die Kenntnis, dass statisch viele Abzweigungen zusammengeführt werden sollen, ist jedoch kein Nachteil gegenüber Workflow-Systemen, denn genau auf diese statische Weise wird das Modell für Workflows ebenfalls entworfen.

Eine entwicklungstechnische Einschränkung gegenüber Workflow-Netzen besteht darin, dass in Workflow-Netzen die Möglichkeit besteht, einen Zweig enden zu lassen, bevor der vorgesehene Endzustand erreicht wird. Hier muss man stets daran denken, dass die Ausführung nicht synchron abläuft und man deswegen nicht unterscheiden kann, ob ein Agent auf irgend ein Ereignis im System wartet oder ob er seine Arbeit als beendet sieht. Diese Zustände müssen aber klar unterschieden werden, um über die Beendigung des Zweiges zu entscheiden. In manchen Workflow-Engines kann diese Situation implizit abgehandelt werden und die Kontrolle beim Beenden automatisch an den *AND*-Join übergeben werden. Bei der Simulation muss man den Fall des vorzeitlichen Beendens stets erkennen können und explizit behandeln.

Die Abbildung 9.3 zeigt eine Modifikation der parallelen Verzweigung, um einen *AND*-Join machen zu können. Um die Abbildung etwas übersichtlicher zu gestalten, wurden Kreissymbole für Agenten weggelassen, die in den Beispielen zuvor noch explizit angegeben worden sind. Stattdessen wird nur der Informationsfluss mittels nichtgestrichelten Pfeilen angedeutet.

Im Teil 9.1.2 wurde davon ausgegangen, dass ein Join nicht nötig ist, was zunächst völlig legitim ist. Möchte man aber die Zweige zählen, konstuiert man mit dem Beschreiben von R_{in} eine exklusive Instanz der Ressource R_Q, die dem

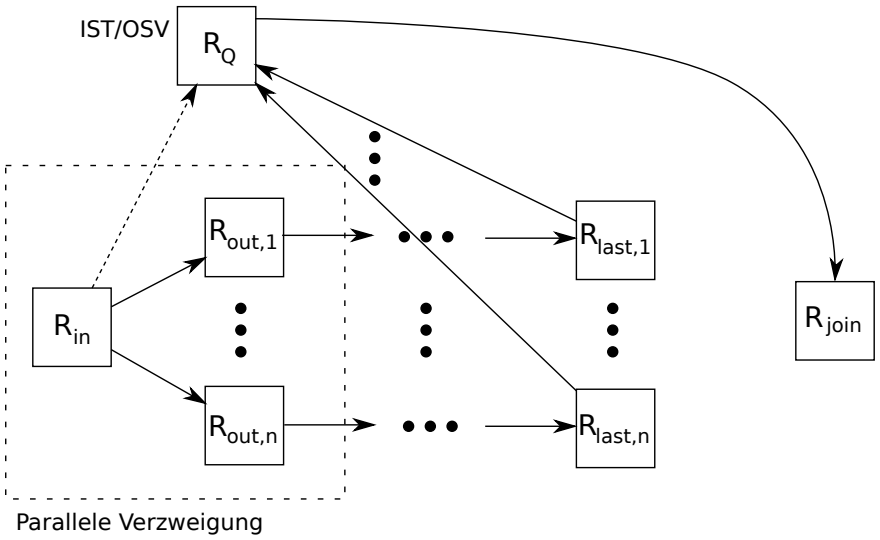

Abbildung 9.3: Simulation eines AND*-Join*

Warteschlangentyp IST/OSV entspricht. Zu dieser Zeit muss bereits klar sein, wie viele Zweige von R_{in} ausgehen und bei der Konstruktion von R_Q mitangegeben werden. Der gestrichelte Pfeil deutet die Instanziierung der Ressource R_Q von R_{in} an.

Die Agenten, die laut Abbildung 9.3 die Meldung über die Ankunft bei $R_{\text{last,i}}$ $(1, \ldots, i, \ldots, n)$ an R_Q übergeben sollen, müssen die dynamisch erzeugte Ressource R_Q über eine Angabe ihrer *URI* finden. Es ist nicht erlaubt, aus der Instanzidentifikation die *URI* abzuleiten, weil das die Bedeutungsneutralität der *URI*s verletzen würde (siehe Teil 5.2.2). Da jedoch jede Ressource $R_{\text{last,i}}$ zu genau einer Ressource R_{in} zugeordnet werden kann, lässt sich nach dem dynamisch konstruierten R_Q anhand der Prozessinstanzidentifikation, die beim Element mitgeführt wird, suchen.

9.1.4 (Multiple) Bedingte Verzweigung

Bei einer *multiplen bedingten Verzweigung* handelt es sich um eine generelle Form der parallelen Verzweigung aus Teil 9.1.2. An dieser Stelle werden jedoch nur einige ausgewählte Zweige verfolgt anstatt aller Zweige. Die Wahl der Zweige wird anhand von bestimmten zur Laufzeit gültigen Kriterien vorgenommen. Sollte der Spezialfall eintreten, dass nur einer der Zweige gewählt wird, verhält sich der Prozess wie bei einer konditionale (*bedingte*) Verzweigung und wird in der Terminologie der Workflow-Netze auch *XOR*-Split genannt. Er wird, im Falle von BPEL, mit Hilfe von `<switch>` (umschlossen von `<flow>`) unterstützt.

Die Simulation muss berücksichtigen, dass für ein späteres Merge, nachvollzuvollziehen sein muss, wie viele Verzweigungen wieder zu einem Zweig verschmolzen werden. Im Endeffekt reicht es also bei der Instanziierung von R_Q von R_{in} (siehe Abbildung 9.3) die Zweige anzugeben, an die die Ausführung weiter gegeben wird.

Wie auch bei der *parallelen Verzweigung* muss dieser Sachverhalt im Zusammenhang mit der zugehörigen Synchronisierung betrachtet werden, denn die Einführung von Parallelverhalten ist in der ressourcenbasierten Architektur nie problematisch. Hier stellt die Synchronisierung das aufwändigere Problem dar, deswegen wird sie nun sofort im Anschluss analysiert.

9.1.5 Synchronisierender Verzweigungszusammenfluss

Der *XOR*-Join ist das Gegenstück zu dem gerade besprochenen *XOR*-Split. Man kann diesen als den allgemeineren Fall eines *AND*-Joins ansehen, bei dem alle Zweige zusammengeführt werden.

Im Falle des *XOR*-Joins müssen die ausgehenden Zweige bei R_Q mitangegeben werden, anhand der festgelegten konditionalen Bedingung, die während der Verzweigung gegolten hat. Dadurch, dass bei der (multiplen) bedingten Verzweigung die aktivierten Zweige in der Ressource R_Q, wie in Abbildung 9.3, festgehalten wurden, ist deren Zahl und Ausführungszustand bekannt und bleibt ebenfalls statisch.

Da R_Q vom Typ *IST/OSV* ist, kann das abschließende, laut Ordnung V erwartete, Element dieser Warteschlange die anschließende Ausführung bei R_{join} anstoßen.

Die Agenten, die R_Q nach dem Erreichen von $R_{\text{last,i}}$ $(1, \ldots, i, \ldots, n)$ beschreiben sollen, finden die *URI* dieser dynamisch erzeugten Ressource, exakt so wie im Falle des *AND*-Joins, indem sie die Ressource R_{in} dazu befragen und die Prozessinstanzidentifikation dazu nutzen, um die Prozessinstanz zuzuordnen.

9.1.6 Explizite Synchronisierung von Aktivitäten

Die *Explizite Synchronisierung von Aktivitäten* ist ein Konstrukt, welches in entkoppelt arbeitenden parallelen Zweigen Kausalität einführt. Die Abbildung 9.4 ist ein Beispiel für das `<link>`-Konstrukt in *BPEL*. Die zwei parallel ausgeführten Zweige mit $V_1, \ldots, V_4$ und $W_1, \ldots, W_4$ sollen unabhängig voneinander arbeiten, mit einer Einschränkung, die durch den dicken Pfeil (so wird in BPEL das `<link>`-Element dargestellt) angegeben wird.

Die Spitze dieses Pfeils zeigt auf W_3 im Workflow-Modell und bedeutet, dass W_3 *erst dann* ausgeführt werden soll, wenn die Ursprungsaktivität des eingehenden

Pfeils abgeschlossen wurde. In der folgenden Abbildung ist es die Aktivität V_3, die vor W_3 abgeschlossen werden muss.

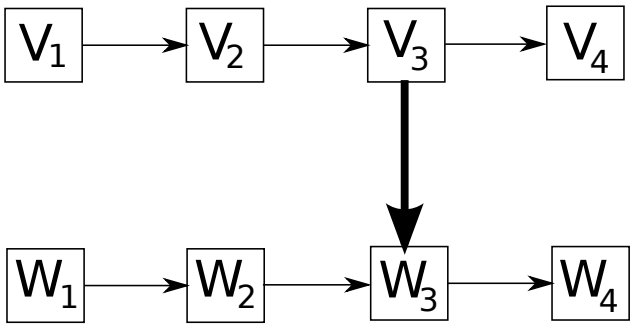

Abbildung 9.4: Synchronisierung mit <link> in BPEL.

Zu beachten ist, dass es mehrere solche Pfeile im Workflow-Modell geben kann. Da es möglich ist, mit <link>-Elementen Deadlocks zu erzeugen, sollen Modellierungswerkzeuge solche nichtzulässigen Kombinationen von <link>s erkennen und vermeiden.

Es wird hier angenommen, dass die Agenten $A_{V_1}, \ldots, A_{V_4}$ die Aktivitäten des Zweigs V und die Agenten $A_{W_1}, \ldots, A_{W_4}$ die des Zweigs W abarbeiten. Dies ist in der Abbildung 9.5 dargestellt. Möchte man <link> in einer ressourcenbasierten Architektur simulieren, dann ist es erforderlich, dass der Agent A_{W_3} den Zustand nach der Ausführung der Vorgänger A_{V_3} und A_{W_2} feststellen muss. Der Ausführungstatus wird in der Abbildung für den Zweig V in den $N_1, \ldots, N_4$ und für den W-Zweig in $M_1, \ldots, M_4$ festgehalten.

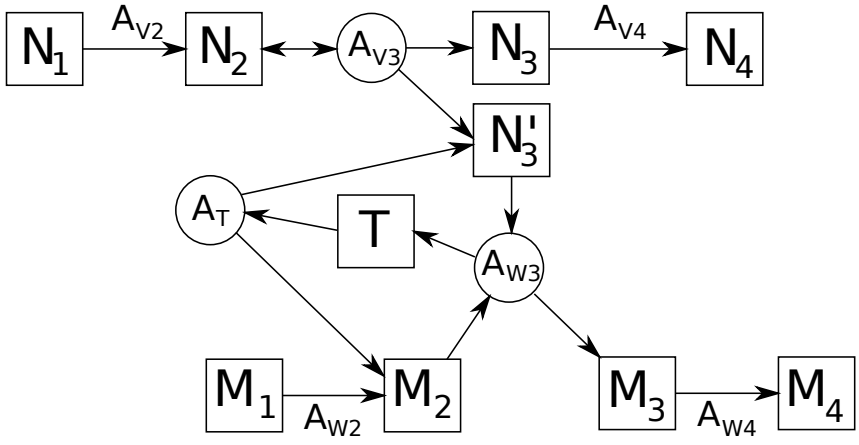

Abbildung 9.5: Simulation der expliziten Synchronisierung von Aktivitäten.

Das erste Problem, welches sich ergibt ist, dass N_3 und M_2 gleichzeitig Ein- und Ausgaberessourcen sind, wenn man die Übertragung der Information abschließend in den beiden Eingaberessourcen bestätigen möchte. Die Abschlussregel für monotone Sequenzen (siehe Teil 8.8.2) kann hier nicht konstruiert werden, denn es ist nur eine Ressource zulässig, die am Ende der Sequenz in der Ein- und Ausgabemenge existieren kann.

Als ein Hilfskonstrukt kommt deswegen ein N_3' als weitere Kontrollressource hinzu, in der der Agent A_{V_3} den Zustand nach der Ausführung neben dem regulären N_3 mitnotiert. Dies ist eine monotone Sequenz, die wie im Agentenprogramm (9.1) aussieht.

$$\begin{aligned} &\text{peek}_{u(N_2)}() : \{d\}, \\ &\text{queue}_{u(N_3)}(\{d\}), \\ &\text{queue}_{u(N_3')}(\{d\}), \\ &\text{delete}_{u(N_2)}(\{d\}) \end{aligned} \tag{9.1}$$

An die Methoden wurden die Ressourcennamen mit Hilfe der Abbildung

$$u : \mathfrak{R} \to \text{URI}, R \mapsto x \text{ mit } x \in U(R)$$

referenziert, weil festgelegt wurde, dass Methoden in der eingeführten Notation, um sie eindeutig zu machen, anhand von *URI*s annotiert werden. Es wird hier angenommen, dass ein x aus der Menge $U(R)$ gewählt wird und, der Einfachheit halber, für alle $y \in U(R)$ es genau das gleiche Verhalten der Methoden gibt.

Der Agent A_{V_4} ist von nun an vollständig abgekoppelt vom übrigen Geschehen. Hingegen wird A_{W_3} erst dann aktiviert, wenn beide Zweige bereit sind und der Einschränkung bezüglich `<link>` genügen. Der Agent wartet dabei auf den Zustand in M_2 und N_3'. Das Agentenprogramm für A_{W_3} ist in (9.2) aufgeführt.

$$\begin{aligned} &\text{peek}_{u(N_3')} : y_{N_3'}, \\ &\text{peek}_{u(M_2)} : y_{M_2}, \\ &\texttt{foreach } d \in \|y_{N_3'}\| \cap \|y_{M_2}\| : \\ &\qquad \text{queue}_{u(M_3)}(\{d\}), \\ &\qquad \text{queue}_{u(T)}(\{d\}) \end{aligned} \tag{9.2}$$

Auf diese Weise besteht die Möglichkeit, die Bestätigung auszulagern und auf das abschließende delete zunächst zu verzichten. Dies zeigt die Abbildung 9.5.

Die Ressource T ist eine Warteschlange des Typs IST/OSN und puffert Bestätigungen. Der Agent A_T tätigt anschließend die Bestätigungsarbeit auf der Ausgabeseite. Die Interpretationsabbildung $\| \cdot \|$ in (9.2) extrahiert die Prozessinstanzidentifikationen in der Antwort in Form einer Menge. Die Mengen der angekommenen Prozessinstanzidentifikationen werden geschnitten und jedes Element d im Schnitt wird an M_3 und T entsprechend weiter gegeben. Die Sequenz ist offensichtlich monoton, weil die Ein- und Ausgaberessourcenmengen disjunkt sind.

Der Agent A_T übernimmt die Arbeit des Aufräumvorgangs für A_{W_3}. Im Verlauf des Agentenprogramms (9.2) wurde auf das abschließende delete auf N_3' und M_2 verzichtet, um eine gültige monotone Sequenz zu erhalten. Dafür wurde in die Warteschlange T das Element d eigefügt, das die Prozessidentifikation beeinhaltet.

Der Agent A_T lässt sich nun folgendermaßen beschreiben:

$$\begin{aligned} &\text{peek}_{u(T)}() : \{d\}, \\ &\text{delete}_{u(N_3')}(), \\ &\text{delete}_{u(M_2)}(), \\ &\text{delete}_{u(T)}(\{d\}) \end{aligned} \tag{9.3}$$

Die Sequenz (9.3) ist monoton laut der Abschlussregel im Teil 8.8. Es gibt hier nur eine Überschneidung der Ein- und Ausgabemenge der Ressourcen und diese wird lediglich am Ende verwendet.

Die übrigen Agenten $A_{V_2}, A_{V_4}, A_{W_2}$ und A_{W_4} sind hier nicht explizit beschrieben worden. Der einfache Pfeil kennzeichnet in der Abbildung 9.5 eine einfache monotone Sequenz zur Übertragung des Datums d (bestehend aus peek, queue, delete).

9.1.7 Diskriminator

Der Diskriminator verhält sich gleich wie die Sequenz aus Teil 9.1.1 in einer ressourcenbasierten Architektur. Das Verhalten in Agentengruppen beruht auf parallelisierter Bearbeitung einer bestimmten Aufgabe. Der erste Agent, der hier ein Ergebnis liefert und das in eine monotoniegetriebene Warteschlange wie *IST/ON* oder *ISV/ON* einreiht, reicht auf der Ausgabeseite die Elemente an eine angekoppelte Bearbeitung.

Die Sequenz unterscheidet sich hier dadurch, dass in der Anwendung lediglich eine Agentengruppe das Ergebnis liefert. Der Diskriminator erlaubt im Prinzip den Eingang von Ergebnissen aus mehreren Quellen (also mehreren Agentengruppen). Bei verschiedenen Agentengruppen ist es im Allgemeinen schwierig

das Kriterium T einzusetzen, weil Ergebnisse aus verschiedenen Quellen oft die gleiche Bedeutung haben, aber nicht vollkommen identisch sein müssen. Die Ergebnisse aus unterschiedlichen Quellen führen unter Umständen zur Bildung von anders konzipierten Elementen, weil die Agenten einer anderen Agentengruppe auf anderen Eingaben operieren können. Eine genauere Äquivalenzbeziehung für Elemente lässt sich dann mit *ISV/ON* erreichen.

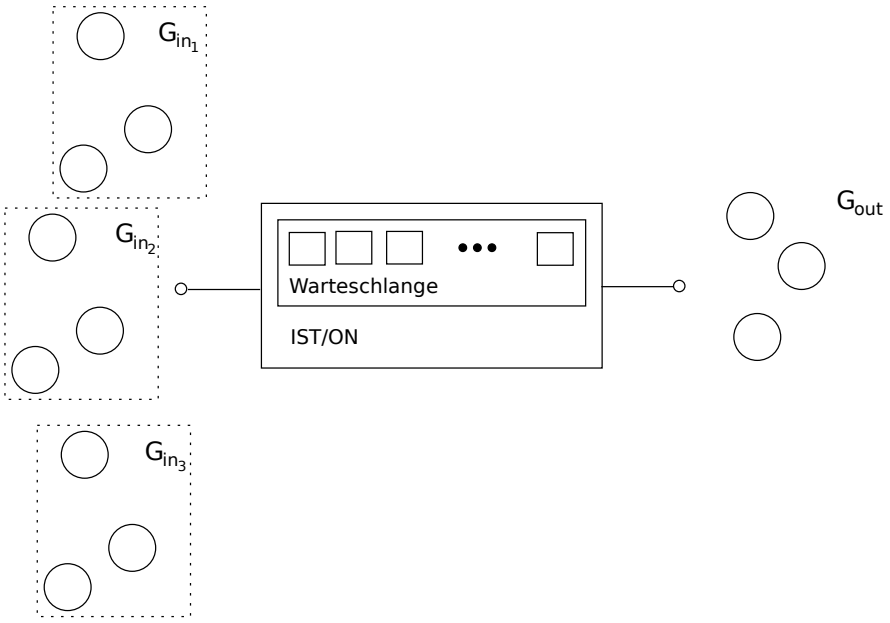

Abbildung 9.6: Der Diskriminator mit mehreren Agentengruppen auf der Eingabeseite.

Das Konzept des Diskriminators ist in der ressourcenbasierten Architektur noch etwas flexibler. Hier werden nämlich auch Ergebnisse aus nicht fest vorgegebenen Quellen mitberücksichtigt. Eine Warteschlange hat keine feste Zahl von möglichen Eingabeinstanzen. In Workflow-Netzen ist diese Zahl vorgeplant und statisch durch das Modell vorgegeben. Außerdem verzichten die meisten Implementierungen (auch BPEL) auf den Diskriminator aufgrund von Komplexität im Zusammenhang mit Schleifenkonstrukten und syntaktischen Problemen [vdAtHKB03] (siehe: Pattern 9 unter „Problem“). Für die ressourcenbasierte Architektur spielen solche Wechselwirkungen zwischen Kontrollmustern und feste syntaktische Beschreibungen keine Rolle. Außerdem ist der Diskriminator ein Muster, welches sich für die Organisation der redundanten Ausführung bei Agentengruppen ausgezeichnet eignet.

9.1.8 Parallele Verzahnung von Aktivitäten

Dieses Muster wird auch „interleaved parallel routing“ genannt und sichert die exklusive Zuteilung von Laufzeit für Aktivitäten in parallelen Zweigen. Es kann recht simpel simuliert werden, indem eine Warteschlange des Typs *IST/OSN* verwendet wird, die zentral zur Verwaltung der Laufzeit der betroffenen parallelen Zweige eingesetzt wird.

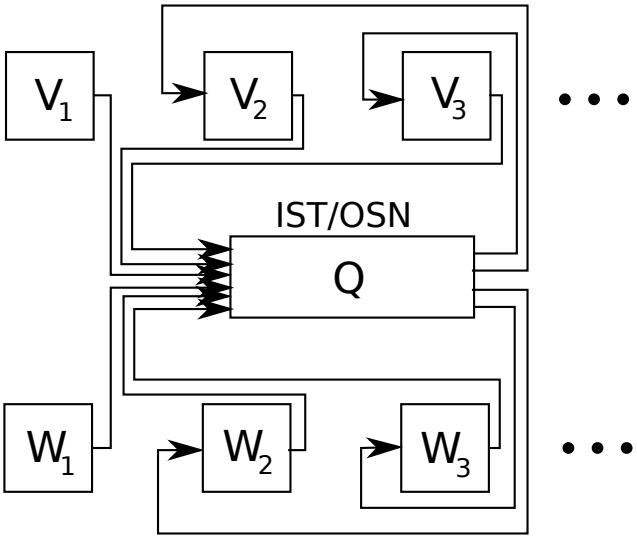

Abbildung 9.7: Exklusive Ausführung in parallelen Zweigen

Die Abbildung 9.7 illustriert die Simulation in einer ressourcenbasierten Architektur. Die Ressourcen V_i und W_j gehören zu zwei verschiedenen Ausführungszweigen, die exklusiv ausgeführt werden sollen. Eine Aktivität, die fertig ist, übergibt die Referenz auf seinen Nachfolger an die Warteschlange Q in der Mitte der Abbildung. Durch die strenge Monotonie auf N auf der Ausgangsseite wird zugesichert, dass die Referenzen geordnet sind nach der Ordnung bei der Ankunft.

Auf der Ausgabeseite von Q sind die Agenten wie in (9.4) programmiert. Sie weisen der Ressource Laufzeit zu, die in der Warteschlange Q als nächstes Element geliefert wird. Die Referenz dieser Ressource ist hier v. Dabei darf v nicht gelöscht werden, weil sonst ein anderes Element zur Ausführung angestoßen wird, bevor die Aktivität, die an v gekoppelt ist, beendet wird. Was an die Ressource mit Referenz v übergeben wird, ist hier nicht weiter wichtig. Um jedoch Prozessinstanzen in angekoppelten Systemen zu unterscheiden, sollte hier die Prozessidentifikation mitangegeben werden.

$$\begin{aligned} &\text{peek}_Q() : \{v\}, \\ &\text{write}_v(\{\}) \end{aligned} \tag{9.4}$$

Die Agenten auf der Eingabeseite der Warteschlange operieren wie im Agentenprogramm (9.5). Ist die Aktivität, die mit u assoziert ist, beendet, dann wird das Lesen mit read_u erlaubt und liefert den Nachfolger u' im Zweig zurück. Die aktuelle Aktivität hinter u wird mit dem Entfernen aus der Warteschlange Q als beendet gemeldet und der Nachfolger u' wird in Q eingereiht.

$$\begin{aligned} &\text{read}_u() : \{u'\}, \\ &\text{delete}_Q(\{u\}), \\ &\text{queue}_Q(\{u'\}), \\ &\text{delete}_u(\{u'\}) \end{aligned} \tag{9.5}$$

Diese Sequenz genügt den Monotoniekriterien. Sie schließt mit dem Beschreiben der Eingabe in der Referenz u. Auffällig ist, dass Q als Warteschlange die Monotonie nicht stört, obwohl sie zwei Mal hintereinander beschrieben wird. Der Fall ist allerdings durch die Abschlussregel abgedeckt, denn Q wird nicht gleichzeitig als Eingabe verwendet.

Bei dieser Simulation ist es etwas unschön, dass jede Agenteninstanz aus der Agentengruppe auf der Ausgabeseite ein write_v ausführen wird, bevor diese feststellen kann, dass es bereits ausgeführt worden ist. Es wurde außerdem die Warteschlange vom Typ IST/OSN gewählt, die nicht mit multiplen Instanzen kooperiert. Das bedeutet, dass diese Warteschlange für jede Prozessinstanz erstellt werden muss. Man kann die Multiinstanzfähigkeit mit dem Warteschlangentyp IST/ON realisieren, jedoch müsste dazu die Ressource spezialisierte Methoden anbieten, die es dem Agenten erlauben, nach Prozessinstanzen zu filtern.

Um den Sachverhalt etwas einfacher darzustellen, wurden hier generische Ressourcen für die Steuerung in den parallelen Zweigen benutzt. Sind mehrere Prozessinstanzen im Spiel, müssten die Ressourcen mit mehr Logik ausgestattet werden. Es würde sich hier im praktischen Einsatz ein Warteschlangentyp empfehlen, der einen Hash als unterliegende Datenstruktur hat und freie Auswahl auf der Ausgabeseite erlaubt. IST/ON wäre dafür am besten geeignet.

9.1.9 Abschließende Bemerkungen

In diesem Unterkapitel wurden typische Workflow-Kontrollflussanweisungen untersucht und es wurde gezeigt, dass man diese mit Hilfe von ressourcenba-

sierten Architekturen simulieren kann. Betrachtet man die Simulation der Kontrollkonstrukte genau, stellt man fest, dass die Synchronisierung aufwändiger ist als die Erzeugung oder Fortführung paralleler Funktionalität. Dies ist natürlich zu erwarten, in einem System, welches Parallelisierung bevorzugt, weil der Aufwand darin steckt, die Daten, die logisch zusammen gehören, zu synchronisieren.

Einige erweiterte Workflow-Muster wurden hier nicht weiter betrachtet. Dazu gehören unter anderem: Iteration oder „deferred choice“. Ein Beispiel für eine komplexere Iteration in Form von einer Rekursion wird im Teil 10.4 gegeben. Das Konstrukt der „deferred choice“ ist, dadurch dass die Ressourcen nicht in einer festen „Verdrahtung“ modelliert werden, kein Problem. In einer ressourcenbasierten Architektur werden Eingaben nicht eingehend auf feste Eingabeagenten beschränkt. Es ist möglich Ergebnisse zu akzeptieren, die „Ad-Hoc“ erscheinen. In BPEL wird eine hier eine externe Entscheidung in Form eines Ereignisses einbezogen, was von `<pick>` realisiert wird.

Des Weiteren führen Workflow-Sprachen die Wiederverwendbarkeit eines Workflows als Aktivität an. Dieses Konstrukt lässt sich als Unterroutine betrachten. Es ist zwar nicht nötig nachzuweisen, dass wiederverwendbare Teile unterstützt werden, aber es sollte klar sein, dass man Systeme in der ressourcenbasierten Architektur dafür gebrauchen kann, um komplexere Berechnungen durchzuführen. Andererseits, lässt sich ebenfalls mit einer Ressource, die den Keller als Datenstruktur hat, ein Aufrufkeller aufbauen, der für eine Prozessinstanz die Ineinanderverschachtelung von Aufrufen modelliert. Dies ist ebenfalls im Beispiel der Rekursion im Teil 10.4 genauer gezeigt.

Eines von den Konstrukten, das hier nicht vorgestellt worden ist, ist „multi merge“, welches ein erweitertes Muster ist. Es beläst die Ausführung nach einer Vereinigung der Ausführungszweige mehrfach weiter ablaufen. Die nachfolgenden Aktivitäten werden so oft ausgeführt wie es Zweige gibt. Bei einigen Workflow-Sprachen, wie zum Beispiel BPEL, wird dieses Muster nicht angeboten. Das hat den Grund, dass das Muster im Zusammenhang mit Iterationen zu Problemen führen kann. Bei der Simulation des „multi merge“ kann man im Prinzip jedem Element eine zweite Identifikation geben, die mit Hilfe von N (der Elementnummer) im jeweiligen verzweigenden Konstrukt in das Element geschrieben wird. Das führt dazu, dass man Prozessinstanzen verfolgen kann und dass die Ordnung T korrekt genutzt werden kann, um Vervielfachungen an gleichen Ressourcen noch zusätzlich nach einer Verzweigung individuell zu betrachten.

Die Verfolgung von Instanzen mit Hilfe von Ressourcen macht die Simulation weniger elegant. Man muss hier aber nicht enttäuscht reagieren, denn dies ist ein Konzept, welches Workflows brauchen und die ressourcenbasierten Architektu-

ren und die darin funktionierende Systeme nicht. Eine andere Möglichkeit wäre, die gesamte Simulationsstruktur für genau eine Prozessinstanz zu instanziieren. Dies klingt vielleicht ungewöhnlich und aufwändig, ist aber ein Weg der durchaus üblich ist. Referenzen auf Ressourcen geben Agenten Hinweise zur Orientierung, wie man von Ressource zu Ressource an die gesuchten Elemente gelangt. Diese Lösung wurde hier vernachlässigt, um die Beschreibung des Verhaltens der Agenten (die Agentenprogramme) kurz zu halten.

Abschließend muss man darauf hinweisen, dass anders als bei Workflows, die Steuerung hier mit Hilfe von Daten oder Ressourcen erledigt wird. Dies zwingt den Entwickler von Prozessen etwas anders zu denken, indem der Prozessstatus stets gesichert wird. Auf diese Weise wird der Prozess konsistent und aktuell gehalten und hängt nicht von der Stabilität des Ausführungssystems ab (also den Agenten). Dieses kann komplett versagen und alle Daten, die hier in Ressourcen bis zu diesem Zeitpunkt festgehalten wurden, werden wieder verfügbar sein. Vorausgesetzt natürlich, dass die unterliegenden Ressourcenspeicher nicht gleichzeitig versagen, was wiederum aber viel einfacher abzusichern ist.

9.2 Petri-Netz

Ein *Petri-Netz* besteht aus *Stellen* und *Transitionen* [Pet62]. Es ist sofort zu sehen, dass Ressourcen hier die Rolle der Stellen übernehmen sollten, weil sie Zustände halten müssen, und dass Agenten für das Schaltverhalten zuständig sind und damit die Transitionen repräsentieren. Ein Problem ist, dass das Schaltverhalten bei einem Agent nicht als lokal betrachtet werden kann. Ein Agent muss stets beachten, ob ein anderer eine Marke gleichzeitig konsumiert und im Konflikt steht.

Im Teil 8.6 wurde erklärt, dass es bei der Steuerung nur eine abschließende Aktion geben kann, die für die konsistente Fortführung von Prozessabläufen sorgt. Petri-Netze sind wegen der Art und Weise, wie ihr Schaltverhalten definiert ist, aus den hier aufgeführten Gründen, schwierig ins Konzept der Agenten und Ressourcen zu übertragen.

Es ist klar, dass Fälle, die in den Abbildungen 9.8 und 9.9 dargestellt sind, handhabbar sind, weil es dort nur genau eine Kontrollstelle für Agenten gibt, die an der abschließenden Aktion beteiligt ist. Diese beiden Fälle sind äquivalent zu der Behandlung der Kontrolle bei einem AND-Join (9.1.3) und bei der parallelen Verzweigung (Teil 9.1.2). Der schwierige Fall entsteht, wenn Stellen auf der Eingabeseite in Konflikt zueinander stehen und der nächste Schritt exklusiv entschieden werden muss. Diesen Fall illustriert die Abbildung 9.10.

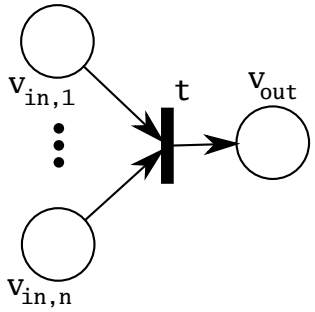

Abbildung 9.8: Kontrollstelle v_{out}

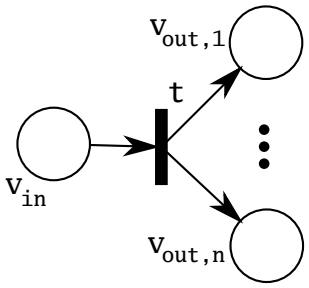

Abbildung 9.9: Kontrollstelle v_{in}

Bei den hier besprochenen Petri-Netzen wird davon ausgegangen, dass die Struktur statisch beschrieben ist und sich nicht ändert. In diesem Fall gilt nämlich, dass es dann ebenfalls möglich ist, eine Kontrollstelle festzulegen und eine monotone Sequenz zur Übertragung, so anzupassen, dass diese Stelle zuletzt adressiert wird. Um diesen Sachverhalt genauer zu betrachten, muss man zuvor weitere Überlegungen anstellen bezüglich des Schaltverhaltens bei Konfliktsituationen.

9.2.1 Konfliktmengen

Der Grund warum das Verhalten interessant ist, ist die Tatsache, dass man gerne die Schaltvorgänge innerhalb eines Petri-Netzes möglichst gut voneinander entkoppeln möchte. Dies führt dazu, dass ein Agent nicht das ganze Netz kennen muss und zudem nicht das ganze Netz sperren braucht, um einen Schaltvorgang durchzuführen. Typischerweise werden Transaktionen verwendet, um exklusives Verhalten zu garantieren und Race-Conditions zu vermeiden, weil es um mehrere Ressourcen und mehrere Agenten geht. Die Transaktionen sollen dabei aber nur beschränkte Mengen von Ressourcen behandeln und nicht mit den Transaktionen für das komplette Petri-Netz vermischt werden, was zu ei-

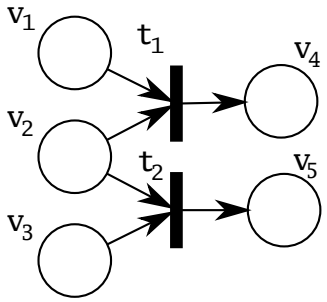

Abbildung 9.10: Stelle v_2 steht im Konflikt

nem „global Lock“ führen und das gesamte Petri-Netz zentralisieren würde (siehe Teil 3.13).

Die *Konfliktmenge* ist die Vereinigung aller Stellen im Vorbereich der Transitionen, die, bei einer beliebigen Markierung, miteinander in Konflikt stehen könnten. Die *Konfliktmenge* beschreibt den Gültigkeitsbereich für eine Transaktion zum Schalten im Petri-Netz, für den genau eine Ressource für die Kontrolle der Sperrungen, anhand von Marken, zuständig sein kann (*Konfliktmengenressource*). Es kann natürlich vorkommen, dass zwei Transitionen, die in der gleichen Konfliktmenge sind, gleichzeitig schalten können, aber dies kann und muss die zugehörige Konfliktmengenressource entscheiden, indem sie die Marken untersucht, die der Agent in der Rolle der Transition verlangt.

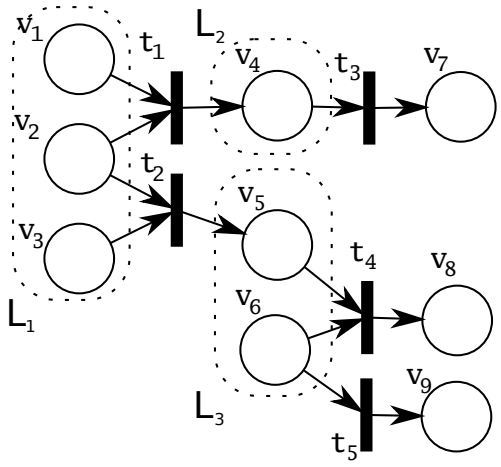

Abbildung 9.11: Beispiel mit Konfliktmengen L_1, L_2 und L_3

In Abbildung 9.11 sieht man ein Beispiel mit drei Konfliktmengen L_1, L_2 und L_3. Das bedeutet, dass wenn man einen Schaltvorgang mit einer Ressourcen aus der Menge L_1 durchführen möchte, muss man *alle* in L_1 liegenden Ressourcen in

Betracht ziehen, wenn es um konkurrierende Transaktionen geht. Andere Ressourcen muss man nicht weiter betrachten. Pro Konfliktmenge braucht man auf diese Weise lediglich eine Ressource zur Verwaltung von Transaktionskollisionen. Wie man so eine Verwaltung realisiert, wird nachfolgend im Teil 9.2.2 gezeigt. Warum es nicht ausreicht, die nur die Ressource zu verwalten, die direkt im Konflikt steht (zum Beispiel $v_2 \in L_1$), wird später klar, wenn die Konfliktmengen den Schaltvorgang bei Petri-Netzen simulieren.

Zunächst wird jedoch ein Algorithmus vorstellt, der die Konfliktmenge L für eine Transition t_0 bestimmt (siehe dazu (9.6)). Es wird hier die, für die Petri-Netze übliche, Notation verwendet.

$$
\begin{aligned}
&\textbf{func } km\ (\ t_0 \in T\ ) : \\
&\quad L := \emptyset \\
&\quad G := \{t_0\} \\
&\quad \textbf{do} : \\
&\qquad L' := L \\
&\qquad \forall t \in G : L' := L' \cup {\bullet t} \\
&\qquad \forall v \in L', t' \in T, W(v,t') > 0 : G := G \cup \{t'\} \\
&\quad \textbf{while } L \neq L' \\
&\quad \textbf{return } L
\end{aligned}
\tag{9.6}
$$

In Worten erklärt erweitert der Algorithmus die Menge $L \subseteq S'$ bis diese nicht mehr wächst, also bis ein Fixpunkt erreicht wird. $S' \subseteq S$ (S enthält alle Stellen im Petri-Netz) ist dabei die Menge aller Stellen im Petri-Netz, die im Vorbereich aller Transitionen in T sind. Im ungünstigsten Fall kann $L = S'$ werden, die minimale Konstruktion von L umfasst den Vorbereich ${\bullet t_0}$. Die Menge $G \subseteq T$ ist hier eine Hilfsmenge, die alle bis jetzt betrachteten Transitionen festhält. Das Ergebnis des Algorithmus ist am Ende in der Menge L. W(v,t) ist 1, wenn eine Verbindung von v nach t existiert, sonst 0.

- man nimmt den Vorbereich in die Konfliktmenge L als Start
- man schaut sich ausgehende Kanten von L an:
  - die Vorbereiche der Transitionen zu denen die Kanten aus L führen schließt man in die Menge mit ein
- dies wiederholt man, bis man einen Fixpunkt erreicht hat (also bis L nicht mehr wächst)

Der Algorithmus hält offensichtlich, denn entweder wächst die Menge L im nächsten Schritt, oder die Berechnung bricht ab. Wenn sie nicht abbricht, dann gilt $L = S'$.□

Überdeckung von S' mit Konfliktmengen

Jede $v_i \in S'$ gehört zu mindestens einer Konfliktmenge L_i. Es gilt laut Voraussetzung, weil v_i im Vorbereich $\bullet t$ von mindestens einer Transition t ist, für die der Algorithmus aufgerufen werden kann.□

Eindeutigkeit der Zuordnung in Konfliktmengen

Sei $v_i \in L_1$ und $v_i \in L_2$ eine Stelle, die in zwei Konfliktmengen vorkommt, also gilt auch $v_i \in S'$. Dann gibt eine Transition t_i, sodass $v_i \in \bullet t_i$ und eine weitere Transition t_j mit $t_i \neq t_j$ und $v_i \in \bullet t_j$. Der Algorithmus in (9.6) wird für $v_i \in L_1$ und $v_i \in L_2$ sowohl t_i als auch t_j in die Menge G aufnehmen.

Dies bedeutet allerdings, dass der Algorithmus für $t_0 = t_i$ und $t_0 = t_j$ die gleiche Menge $L = L_1 = L_2$ berechnet, in der v_i enthalten ist.□

Schnitt von Konfliktmengen

Die Eindeutigkeit der Zuordnung und die Überdeckung von S' impliziert auch, dass der Schnitt $L_1 \cap L_2$ von je zwei Konfliktmengen L_1 und L_2, mit $L_1 \neq L_2$, leer ist.

Dies bedeutet, dass insbesondere jede Stelle, die im Konflikt steht, weil sie im Vorbereich von mindestens zwei Transitionen steht, genau zu einer Konfliktmenge zugeordnet ist.

Genereller gilt, dass jede Stelle in S' zu einer eindeutigen Konfliktmenge zugeordnet wird. Dies ist wichtig, weil es nicht nur um die Parallelisierung von Agenten geht, die in verschiedenen Agentengruppen konkurrieren, sondern auch in der gleichen Agentengruppe. Die Konflikte der Agenten gleicher Agentengruppe umfassen nämlich nicht nur die Ressourcen (hier Stellen in S'), die im Konflikt zu anderen Agentengruppen stehen. Eine Optimierung auf die Menge der Stellen, die mindestens den Ausgangsgrad 2 haben, liegt zwar nahe, ist aber aus den eben erwähnten Gründen falsch.

Spezialfall am Beispiel der Philosophen

Die Konfliktmenge im Falle der „Dining Philosophers“ [Hoa04] umfasst alle Stellen (in der Abbildung 9.12 sind es die Gabeln $v_1, \ldots, v_4$). Der Kern des Problems besteht darin, dass alle Transitionen $t_1 \ldots t_4$ die Stellen miteinander so verbinden, dass der Algorithmus für Konfliktmengen alle Gabeln auf dem Tisch in eine und die einzige Konfliktmenge $L = \{v_1, v_2, v_3, v_4\}$ hinzufügt.

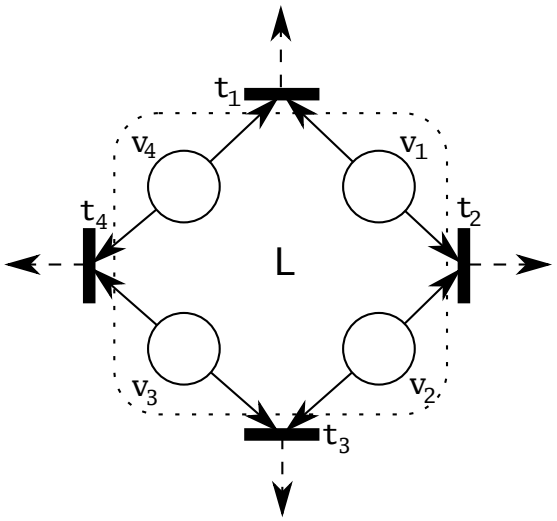

Abbildung 9.12: Spezialfall für L bei den „Dining Philosophers“.

Dies bedeutet für die Agenten, die anstelle der Transitionen entscheiden sollen, ob sie schalten oder nicht, eine Ressource brauchen, die die Übersicht über alle vier Stellen, die in L aufgeführt sind, haben muss. Es ist dabei natürlich trotzdem zulässig, dass zwei Transaktionen, zum Beispiel beim Schalten der Transitionen t_1 und t_3 (oder t_2 und t_4), möglich wären.

9.2.2 Schalten in beliebigen Petri-Netzen

Die Konfliktmengen und der km-Algorithmus im Teil 9.6 können nun dazu verwendet werden, um ein statisches Petri-Netz mit beliebiger Struktur zu simulieren. Das Petri-Netz muss deswegen statisch sein, weil die Konfliktmengen sich dann während der Laufzeit nicht ändern.

Für jede berechnete Konfliktmenge $L_1, \ldots, L_m$ mit $m \geq 0$ muss zusätzlich eine Konfliktmengenressource R_{L_i} angelegt werden, die Ordnung festlegt und Kollisionen erkennt im Falle von mehreren Transaktionen in einer Konfliktmenge L_i. Die Anfragen zum Schalten unter den Transaktionen kommen von den Agenten, die die Rolle der Transitionen übernehmen. Jede Ressource im Vorbereich eines Agenten, der eine Transition simuliert, ist in der gleichen Konfliktmenge.

Der Agent muss sich deswegen nur mit genau einer Konfliktmengenressource abstimmen, ob eine Transition schalten kann.

Aufbau einer Konfliktmengenressource

Die *Konfliktmengenressource* R_{L_i} entspricht dem Warteschlangentyp ISV/ON (siehe Teil 7.8.6). R_{L_i} empfängt dabei Mengen E_j von zu sperrenden, in den Eingangsressourcen liegenden, Marken, für die gilt $E_j \subseteq L_i$ und eine festgelegte Identifikation der Transition, die der Agent simuliert. Es ist dabei anzumerken, dass hier die Zustandslosigkeit beibeihalten wird, weil Agenten einer Agentengruppe, die das Verhalten einer Transition simuliert, nicht diskriminiert werden. Den Marken werden außerdem eindeutige Identifikationen gegeben, damit die Warteschlange für die Konfliktmengenressource sie unterscheiden kann.

Die strenge Monotonie auf dem Eingangsverhalten verlangt dabei, dass R_{L_i} nie zwei Mengen $E_{j'}, E_{j''} \subseteq L_i$ speichert, für die gilt $E_{j'} \cap E_{j''} \neq \emptyset$. Die Ausgabeseite der Warteschlange erlaubt wahlfreien Zugriff auf die Elemente (die Mengen), um paralleles Verhalten zu fördern und damit den Ablauf von je zwei nichtkollidierenden Sperren innerhalb einer Konfliktmenge L_i zu ermöglichen. Der Agent kann dabei anhand der Identifikation der Transition die Wahl des gespeicherten Elements E_j entsprechend einschränken.

Aufbau einer Stellenressource

Eine Stelle wird durch eine Warteschlange des Typs *IST/ON* (siehe 7.8.8) simuliert. Hiermit wird gesichert, dass identische Marken, die eine Agentengruppe in die Stelle einbringen möchte, still konsumiert werden. Die Ausgangsseite erlaubt der Agenten wahlfreien Zugriff auf alle gespeicherten Marken, die im Gegensatz zu Petri-Netzen in der Simulation eindeutig identifizierbar sind.

Empfängt eine Stellenressource eine Marke, dann generiert sie nach der Identitätsfeststellung eine andere Identifikation (also eine andere Marke). Hiermit wird das Konsumieren der alten Marken simuliert und zugesichert, dass Marken mit gleicher Identifikation nicht an zwei verschiedenen Stellen im Petri-Netz auftauchen. Dank eindeutiger URIs der Ressourcen und der internen Nummerierung, ist die Konstruktion von eindeutigen Identifikationen ganz einfach.

Agentenverhalten

Das Gegenstück zu der Gestaltung der Ressourcen ist das Verhalten der Agenten. Wie im Teil 8.6 eingeführt, soll hier auch eine monotone Sequenz als Werkzeug

benutzt werden, um das Schaltverhalten in Agentengruppen zu spezifizieren.

Das folgende Agentenprogramm in (9.7) beschreibt das lokale Verhalten der Agentengruppe, die die Transition t_0 simuliert.

$$
\begin{aligned}
&V = \emptyset \\
&\forall v \in \bullet t_0 : \texttt{peek}_v() : \{x_{v_1}, \ldots, x_{v_i}, \ldots, x_{v_n}\} \\
&\qquad \text{für beliebiges } j, 1 \leq j \leq n : V = V \cup \{x_{v_j}\} \\
&\qquad \text{gilt für ein } v : n = 0, \text{ dann } \texttt{RESTART} \\
&\texttt{queue}_{L_i}(\ldots, x_{v_j}, \ldots, \mathrm{Id}(t_0)) \\
&\texttt{peek}_{L_i}(\{t_0\}) : \{\ldots, x'_{v_j}, \ldots, \mathrm{Id}(t')\} \text{ oder } \{\} \\
&\text{wenn } \{\} \text{ oder } \mathrm{Id}(t') \neq \mathrm{Id}(t_0) : \texttt{RESTART} \\
&\forall w \in t_0 \bullet : \texttt{queue}_w(\ldots, x'_{v_j}, \ldots, \mathrm{Id}(t_0)) \\
&\forall v \in \bullet t_0 : \texttt{delete}_v(x'_{v_j}) \\
&\texttt{delete}_{L_i}(\ldots, x'_{v_j}, \ldots, \mathrm{Id}(t_0))
\end{aligned}
\tag{9.7}
$$

Diese Lösung besteht aus zwei eigenständigen monotonen Sequenzen. Das Verfahren zur Konstruktion solcher komponierten Sequenzen wurde im Teil 8.9 vorgestellt. Hier endet die erste Sequenz bei $\texttt{queue}_{L_i}$ und wird durch die monotone Warteschlange entkoppelt, sodass der Agent, der bei $\texttt{peek}_{L_i}$ weiter sein Programm vorführt, völlig eigenständig operieren kann.

Im Agentenprogramm werden zunächst im Vorbereich der Transition die Marken zusammengesucht. Pro Stelle v wird hier eine Liste der verfügbaren Marken geliefert. Mittels j wird nichtdeterministisch eine Marke in der Stelle ausgewählt. Sind keine Marken zum Schalten in einem v verfügbar, kann die Transition nicht schalten und wird später noch einmal angestartet.

Konnte eine Auswahl von Marken (hier als $\ldots, x_{v_j}, \ldots$ angegeben) getroffen werden, wird diese in die Transaktionsressource L_i mit der Transitionskennung $\mathrm{Id}(t_0)$ abgespeichert. Auf der Ausgabeseite der Transaktionsressource wird mittels $\mathrm{peek}_{L_i}(\{t_0\})$ eine gefilterte Anfrage gestellt, die alle Sperren die von t_0 gemacht worden sind zurück gibt. Gibt es solche Sperren nicht ($\{\}$), dann ist nichts weiter zu tun und der Agent kann später wieder aktiviert werden. Die Einschränkung $\mathrm{Id}(t') = \mathrm{Id}(t_0)$ ist zwar nicht nötig, weil die Filterung der Anfrage das zusichert, aber bekräftigt den Zusammenhang von t_0 und t'[1].

An diesem Punkt kann die eigentliche Markenübertragung passieren. Für jede Stelle w im Nachbereich wird die Kombination aller in der Transaktion reservier-

[1] Dies wurde nur zum besseren Verständnis gemacht. Die Unterscheidung der Bezeichner illustriert lediglich, dass die Sequenz in zwei eigenständige Teile zerfallen kann.

ten Marken $(\ldots, x'_{v_j}, \ldots)$ gespeichert mitsamt der Transitionskennung $\mathrm{Id}(t_0)$. Erst danach können alle diese Marken einzeln im Vorbereich gelöscht werden. Pro Position im Feld $\ldots, x'_{v_j}, \ldots$ wird ein solches x'_{v_j} dazu verwendet (diese Schreibweise ist stark abgekürzt). Am Ende der Sequenz wird die Transaktion aus der Ressource L_i mit delete_{L_i} gelöscht.

9.3 Schlusswort zu Simulationen

In diesem Kapitel wurde gezeigt, dass ressourcenbasierte Architekturen sowohl die Steuerung der Workflow-Netze als auch der Petri-Netze simulieren können.

Bei den Workflow-Netzen wurden die wichtigsten *Workflow-Patterns* [vdAtHKB03] ausgewählt, die zur Prozesskontrolle dienen. Zu jedem dieser Muster wurde eine Abbildung auf die ressourcenbasierte Architektur gemacht.

Bei den Petri-Netzen wurden die Stellen als Ressourcen eingesetzt und das Schaltverhalten mit Hilfe der Agenten simuliert. Durch die in den vorhergehenden Kapiteln besprochenen Ressourcen- und Protokolleigenschaften wurde gezeigt, dass das Schaltverhalten ausreichend simuliert werden kann, um den Betrieb eines Petri-Netz zu gewährleisten.

Die Petri-Netze sind noch etwas schwieriger zu unterstützen, wenn man die Ausgabe beim Schalten atomar gestalten möchte. Dies ist zwar für die Funktion nicht erforderlich, führt aber zu einem sauberen Konzept. Außer dem vollständig atomaren Schaltverhalten wurde Fairness [Mur89] beim Schalten vernachlässigt. Diese Eigenschaft ist für die Simulation an dieser Stelle ebenfalls nicht nötig und ist thematisch anders aufgestellt. Nur als kurze Erklärung dazu: es wird nicht verlangt, dass Agenten bestimmte Garantien bezüglich Laufzeit und der Häufigkeit von Anfragen erfüllen. Durch die Abwesenheit einer vollständigen Agentengruppe im Konfliktfall ergibt sich zwangsweise ein unfaires Verhalten beim Schalten.

Anhand der hier besprochenen Ergebnisse kann man erkennen, dass ressourcenbasierte Architekturen zwar einfach aufgebaut sind, weil sie vor allem datenbasiert sind, aber sie auch in der Lage sind, vielseitige Anwendungen zu unterstützen. Sie sind nämlich durchaus ausdrucksstark genug, um die herkömmlichen Prozessmodelle simulieren zu können.

Kapitel 10

Experimentelle Auswertung

Ressourcenbasierte Architekturen haben Vorteile, wenn das Standardparadigma der Parallelisierung genutzt wird und die Synchronisierung in wenigen Einzelfällen gemacht werden muss, zum Beispiel abschließend, wie bei Prozessen die mit Hilfe von Kooperation zu einem Gesamtergebnis kommen, die nach längerer Zeit verteilter Arbeit, das Ergebnis erst zusammenfügen sollen. Die Synchronisierung wird nur dann gebraucht, wenn Resultate aus verschiedenen Quellen auf eine vorgegebene Art und Weise vereint werden müssen oder einfach beim Transport von Informationen von System zu System.

Die in diesem Kapitel vorgestellten Experimente wurden mit dem *cgi2c*-Framework (siehe Teil 6) für Ressourcen und Shell- und Web-Skripten für Agenten. Ein Teil wurde auch in der neuen Programmiersprache *Go* [GoL12] gemacht für die sich die vorgestellten Paradigmen und Prinzipien gut gebrauchen lassen.

10.1 Aktionsprotokoll

Einführend wird ein einfacher Anwendungsfall für Informationsausbreitung mit einer Synchronisierenden Funktion vorgestellt. Es handelt sich um das zentralisierte Einsammeln von zu protokollierenden Aktionen von verteilten Systemen. In der Abbildung 10.1 ist der Architekturaufbau dargestellt.

Dazu werden verteilt Systeme $S_1, \ldots, S_4$ betrieben, auf denen Aktionen passieren, die zentral auf einem Protokollsystem P gespeichert werden sollen und zwar in der Reihenfolge, in der sie dort eingegangen sind. Die Ordnung wird mit Hilfe einer zwischenliegenden Warteschlange Q des Typs *IST/OSN* konstruiert. Q akzeptiert auf der Eingabeseite Protokolleinträge, die von den zu protokollierenden System einkommen. Ein Agent, der für P Information überträgt, holt diese von Q geordnet ab.

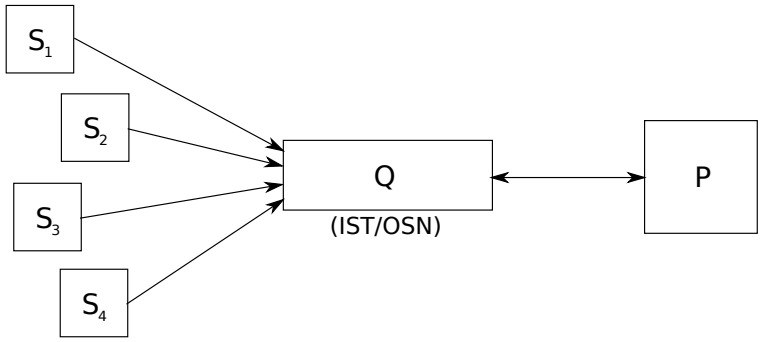

Abbildung 10.1: Beispiel für verteiltes Protokollieren

Es ist darauf zu achten, dass *IST/OSN* Maßnahmen gegen Vervielfachungen von Protokolleinträgen anbietet. Das bedeutet, dass $S_1, \ldots, S_4$ Systeme sind, bei denen Vervielfachungen auftauchen könnten. Eine ähnliche Art von Anwendung wird zum Beispiel von *Syslog* [Ger09] auf Unix-Systemen angeboten, um Systemprotokolle zu verwalten und abzuspeichern.

10.2 Rekordstände am Deich

Ein interessanter Fall für den Einsatz des Warteschlangentyps *IV/OSN* ist ein Deich, welcher durch einen Sensor überwacht wird. Der Endanwendung soll Rekorde der Wasserhöchstände mitgeteilt bekommen und zwar auch dann, wenn der größte zuvor gemeldete Höchstand wieder erreicht wird.

Die Abbildung 10.2 zeigt die Systeme, die an dem Aufbau beteiligt sind.

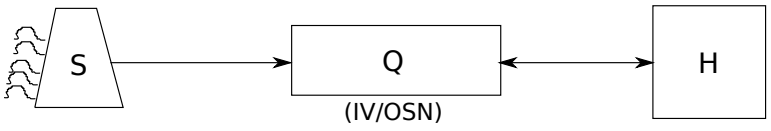

Abbildung 10.2: Beispiel für einen „Rekordsensor“ am Deich

Am Deich ist ein Sensor S installiert, welcher Messwerte an die Warteschlange Q liefert. Q inspiziert die gemeldete Werte (Kriterium V), indem der Wasserstand, mit Hilfe der zugehörigen Abbildung t, extrahiert wird. Wegen einfacher (nicht strenger) Monotonie, werden auch Rekordhöchststände in der Zukunft nachverfolgt. Die aufkommenden Daten werden im System H für zukünftige Analyse festgehalten. In H kann man auch die Dauer der Rekordhöchststände nachvollziehen.

Dieses Beispiel ist auch ein sehr einfaches Szenario für Entkopplung. Falls ein Untersystem bei S oder bei H (genauer: der zugehörige Agent) einzeln ausfällt, kann das jeweils andere Subsystem entkoppelt gut funktionieren.

10.3 Ein persönlicher Agent

Als ein weiteres Beispiel kann das *World Wide Web* verstanden werden, in dem Agenten ganz natürlich für den Informationstransport zuständig sind. Agenten sind Web-Clients, und diese sind nicht nur die Web-Browser, sondern können auch automatisierte Programme sein.

Angenommen man hat im Web einen Cloud-Service, der es einem erlaubt mit einer bestimmten PaaS-API („platform as a service“) Agenten vielfältig zu programmieren, die für persönliche Zwecke eingesetzt werden könnten. Diese Agenten können von Benutzern eingesetzt werden, um Informationen von Web-Seiten einzusammeln und diese an ein separates System zu übergeben, das den Status in Form von Benachrichtigungen per E-Mail täglich zuschickt.

Da Agenten in der Formation der Agentengruppe abstrakt als eine einzelne Agenteninstanz beschrieben werden können, wenn sie die hier in der Arbeit eingeführten Prinzipien verwenden, kann hier auch repräsentativ von einem Agenten ausgegangen werden, der für den Benutzer Arbeit leistet.

Die beteiligten Systeme sind in der Abbildung 10.3 zu sehen und sind in gestrichelten Rechtecken dargestellt.

Oben in der Abbildung sind die zu überwachenden Web-Seiten, die der Benutzer frei auswählen kann. Der *Cloud-Service* stellt für die Überwachung der Web-Seiten Agenten zur Verfügung (A_1 und A_2), diese Übergeben den Status der Webseite an die Zwischenspeicher Q_1 und Q_2. Diese Ressourcen sind sinnvollerweise Warteschlangen des Typs IST/OSN. Damit werden Vervielfachungen der Benachrichtigungen über Veränderung der überwachten Web-Seiten vermieden.

Die Agenten A'_1 und A'_2 sind für die Ausgabeseite der Warteschlangen Q_1 und Q_2 zuständig. Sie laufen ein Mal am Tag und geben den aktuellen Status, der in der Warteschlange vorgehalten wird, an eine Ressource, die den E-Mail-Versand abhandelt. Diese Ressource ist natürlich synchron gekoppelt (symbolisiert durch die dicke Verbindung) an einen SMTP-Server[1].

Nicht weiter betrachtet werden hier die Ressourcen, die benötigt werden, um Benutzer zu den zu überwachenden Ressourcen zuordnen. Sie beinflussen hier das Verhalten der Überwachungsagenten, aber sind größtenteils Ressourcen, auf

[1] ein SMTP-Server nimmt E-Mails zum Versand entgegen und stellt den üblichen Weg dar, wie E-Mails im Internet transportiert werden

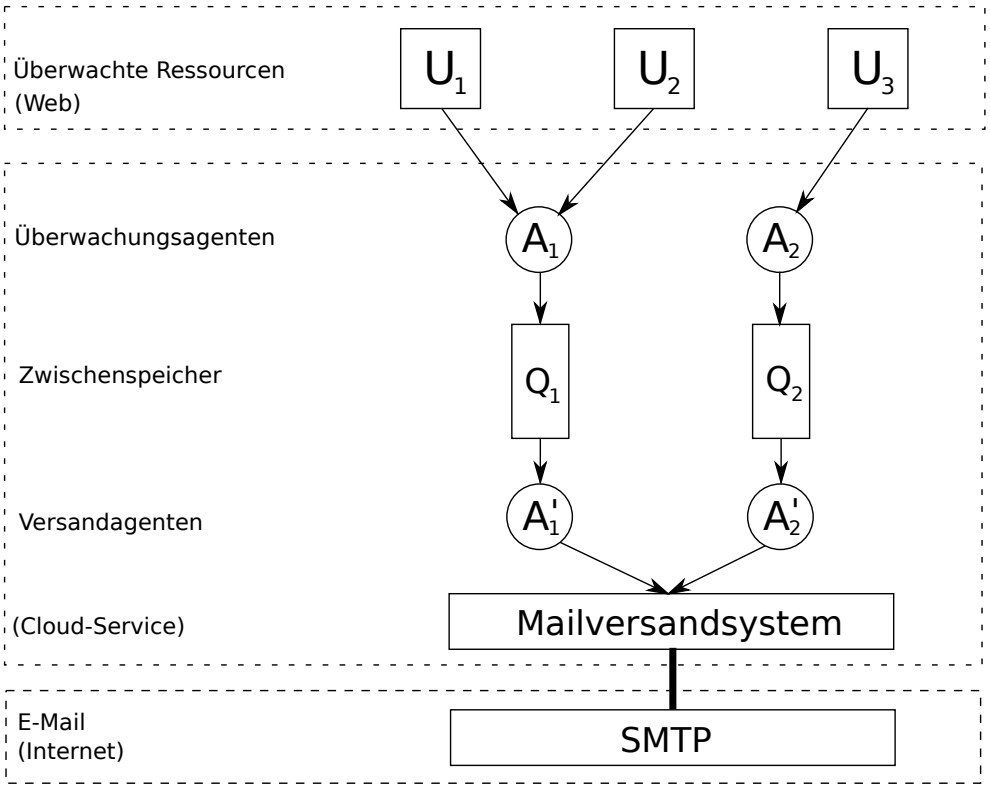

Abbildung 10.3: Beispiel für Web-Seiten-Überwachung mit Agenten

die nur lesend von den Agenten zugegriffen wird. Die Versandagenten könnten ebenfalls an ein Abrechnungssystem angekoppelt werden, das hier auch nicht weiter aufgeführt worden ist.

10.4 Rekursive Probleme

Im Teil 7.10 wurde eine IST/OSN-Warteschlange mit Idempotenz ausgestattet und so modifiziert, dass sie wie ein *Keller* funktioniert. In diesem Beispiel wird nun gezeigt, wie diese Ressource gebraucht werden kann, um die *Fibonacci-Zahlen* zu berechnen.

Natürlich ist die nichtrekursive Formell für die Fibonacci-Folge bekannt. Hier wird sie nicht benutzt, um zu illustrieren, wie die Ressource und die Agenten zusammen funktionieren können, um verteilt ein Problem zu lösen.

Dieses Beispiel wurde in *Go* implementiert, welches mit dem Konzept von *CSP* [Hoa04] [Hoa78] Kommunikation zwischen Prozessen realisiert. Dazu wurden die benötigten Ressourcen als ein eigenständiger Prozess realisiert, der einen Kommandokanal hat und dort per Request-Reply Methoden empfängt.

Die Agenten sind ebenfalls Prozesse, die Methoden auf den Kommandokanälen an die Ressourcen senden und Antworten erhalten. Die ressourceninterne Synchronisierung wird hier mit Hilfe ihrer Kommunikationskanäle realisiert. Jeder Methodenaufruf ist atomar gestaltet und wartet dort bis die Ressource ihn verarbeitet.

Diese Architektur entspricht dem hier behandelten ressourcenbasierten Architekturstil. In Abbildung 10.4 ist eine Skizze des Systems.

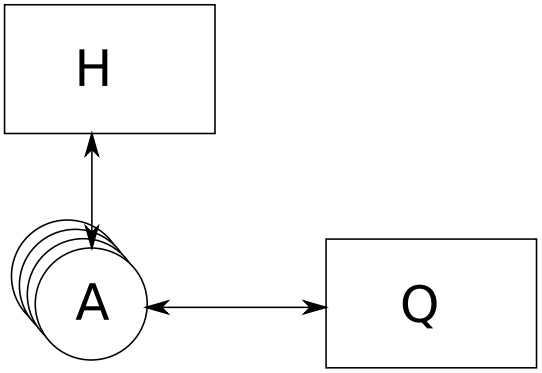

Abbildung 10.4: Beispiel für rekursiven Prozess

Die Ressource Q dient hier als Kontrollressource und steuert die Berechnung der Fibonacci-Folge mit Hilfe von Ordnungen und Monotonie. In Q werden die einzelnen Werte (v) gespeichert, die berechnet werden müssen. Die Methoden der Warteschlange Q sind in (10.1) aufgeführt. Es sind die Methoden der Kellerressource aus Teil 7.10. Die hier aufgeführte Methode delete_Q entspricht dem Verhalten der Methode pop^*_Q.

$$\begin{aligned} &\text{push}^*_Q(\{v\}) \\ &\text{delete}_Q(\{v\}) \\ &\text{peek}_Q() : \{v\} \end{aligned} \tag{10.1}$$

Die zweite Ressource H speichert die bereits berechneten Glieder der Fibonacci-Folge. Die Methoden von H sind in (10.2) aufgeführt, wobei hier die Zustandsänderungen mitangegeben sind. Die erste Methode get_H gibt das i-te Folgenglied w der Fibonacci-Folge an, falls es bereits berechnet wurde. Sollte es noch nicht berechnet worden sein, wird ein entsprechender Fehler zurückgeliefert. put_H ist die zweite Methode und speichert das i-te Folgenglied, welches als w übergeben worden ist.

$$\begin{aligned}
&\text{get}_H(\{i\})/[Z_i \to Z_i] : \texttt{error} \\
&\text{get}_H(\{i\})/[Z_i' \to Z_i'] : \{w\} \\
&\text{put}_H(\{i, w\})/[Z_i \to Z_i'] \\
&\text{put}_H(\{i, w\})/[Z_i' \to Z_i']
\end{aligned} \tag{10.2}$$

Die Ressource ist ebenfalls monoton gestaltet und entspricht dem Verhalten einer Hash-basierten monotonen Warteschlange des Typs IST/ON.

Das Agentenverhalten für A, und seiner Agentengruppe bei der Berechnung der Fibonacci-Folge, läuft ab, wie in (10.3) spezifiziert.

$$\begin{aligned}
&\text{peek}_Q() : \{v\}, \\
&\text{get}_H(\{v\}) : \{w\} \cup \texttt{error}, \\
&\texttt{if (!error)} \\
&\quad \text{delete}_Q(\{v\}), \\
&\quad \texttt{RESTART} \\
&\text{get}_H(\{i-2\}) : \{v'\} \cup \texttt{error}', \\
&\text{get}_H(\{i-1\}) : \{v''\} \cup \texttt{error}'', \\
&\texttt{if (!error}' \texttt{ \&\& !error}'') \\
&\quad \text{put}_H(\{i, v' + v''\}), \\
&\texttt{RESTART}
\end{aligned} \tag{10.3}$$

Die beiden möglichen Ausführungswege mit zustandsändernden Aufrufen haben die Suffixe delete_Q und put_H. Dies sind auch die einzigen Schreibzugriffe im Agentenprogramm für A und somit ist offensichtlich, dass sie monotone Sequenzen ergeben.

Die Agenten A berechnen kooperativ die Fibonacci-Folge für die Zahl v_f, die mit einem initialen Aufruf von $push_Q^*(\{v_f\})$ an die Ressource Q übergeben worden ist.

An diesem Beispiel lässt sich erkennen, dass die Berechnung hier nicht gut parallelisierbar ist. Die Agentengruppe wird hier vor allem die redundante Ausführung gegenüber der verteilten bevorzugen. Es hilft auch in diesem Fall nicht, die Kellerstruktur mit der freien Wahl auszustatten, denn das Problem ist hier so formuliert worden, dass es die Agenten zwingt nacheinander die Folgenglieder zu berechnen. Man kann die nächste Zahl nur dann berechnen, wenn die Vorgänger bereits bekannt sind. Nichtsdestotrotz ist hier in diesem Beispiel illustriert, wie

die Kontrolle von Daten in zwei verschiedene Ressourcen aufgetrennt worden ist, wie monotone Warteschlangen in Zusammenarbeit mit den monotonen Sequenzen eingesetzt werden können. Die Prinzipien, die für Ressourcen und das Agentenverhalten aufgestellt worden sind, bleiben hier erhalten und führen zu einer Lösung. Die eingesetzte Architektur, die auf *Go* und den *CSP* basiert, zeigt auch, dass der eingeführte Architekturstil vielseitiger ist und nicht nur im World Wide Web (auf Basis von *REST*) funktioniert.

10.5 Kooperatives Lösen eines Problems

Dieses Beispiel wurde ebenfalls in *Go* implementiert. Hier geht es um das kooperative Lösen eines Problems in Form eines Prozesses, bei dem klar und einfach umschrieben werden kann wann der Prozess abgeschlossen ist. Das Problem ist ein Puzzle verteilt zu lösen, ohne dass die aktiven Beteiligten miteinander kommunizieren können. Die Verarbeitung des Prozesses kann man in der Abbildung 10.5 sehen.

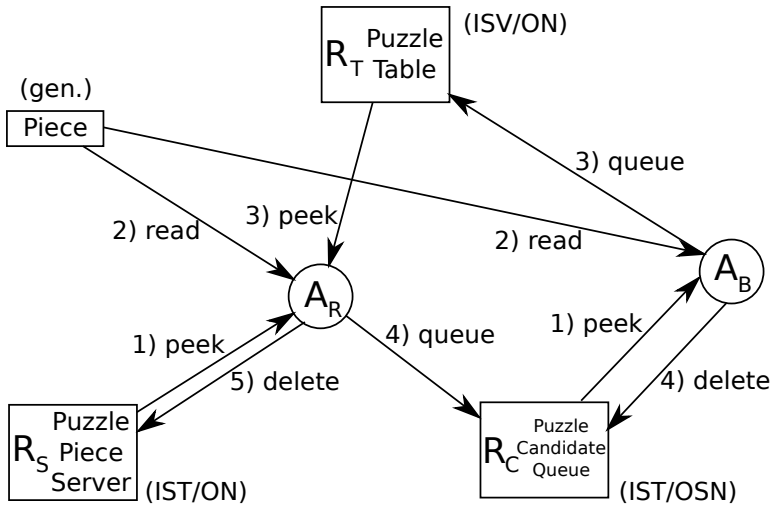

Abbildung 10.5: Beispiel für einen kooperativen Prozess

Jedes Puzzlestück hat in der Zielressource R_T (*Puzzle Table*) einen festen Platz. Ein konstruiertes Hindernis dabei ist, dass ein Agent nicht herausfinden kann, ob das Puzzlestück an eine Stelle passt, ohne es auszuprobieren. Des Weiteren kann ein Puzzlestück nicht in die *Puzzle Table* abgelegt werden, wenn die Teillösung nicht zusammenhängend ist. Dabei sind die Ränder der Puzzlelösung vorgegeben, damit man überhaupt anfangen kann. Weiß ein Agent, dass ein Puzzlestück an eine Stelle passt, kann er lokales Wissen darauf basierend aufbauen, welche Puzzlestücke angrenzend drangesetzt werden können. Wie Puzzlestücke im Zu-

sammenhang mit anderen angrenzenden Puzzlestücken repräsentiert werden, kann man in Abbildung 10.6 sehen.

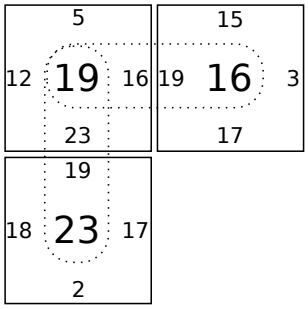

Abbildung 10.6: Repräsentation von Puzzlestücken

Passt das Puzzlestück (in Abbildung 10.6 könnte das Puzzlestück mit Identifikationsnummer 17 unten rechts abgelegt werden), legt der Agent es nicht selbst in die Zielressource, sondern übergibt die Aufgabe an einen Agenten, der auf diese Aufgabe spezialisiert ist. Es gibt hier also zwei Agentengruppen, die man beliebig skalieren kann. Einmal die Rateagenten, die herausfinden können, ob und wo das Puzzle hinpasst und die Bastelagenten, die das Puzzlestück dann endgültig plazieren können. Die Agentengruppen sind hier repräsentativ durch jeweils einen Rateagenten A_R und einen Bastelagenten A_B angegeben.

Die Puzzlestücke sind als generische Ressourcen realisiert und müssen als Daten angefordert werden, bevor sie in der Ressource *Puzzle Table* ausprobiert werden können. Hier sollte der Unterschied zwischen Kontrolle und Daten auffallen und es sollte klar werden, dass ein Cache im Falle der Puzzlestücke Aufrufe sparen kann. Der aktuelle Zustand des Prozesses wird mit Hilfe der Ressource R_S *Puzzle Piece Server* festgehalten. Dort sind alle noch nicht abgelegten Puzzlestücke gespeichert.

Es ist nicht festgelegt, welchen Startzustand es hier gibt. Für die Rateagenten ist es wichtig, dass sie zu jedem Zeitpunkt ihre Arbeit fortführen können. Das bedeutet auch, dass sie den aktuellen Zustand der Lösung stets erkennen müssen. Dies ist hier natürlich sehr einfach, denn der aktuelle Zustand lässt sich anhand der Befüllung des *Puzzle Piece Server*s feststellen. Der Rateagent A_R muss natürlich eine monotone Sequenz benutzen (siehe (10.4)).

$$
\begin{aligned}
&\text{peek}_{u(R_S)}(\{i\}) : \{v\} \text{ (für ein beliebig gewähltes } i) \\
&\text{read}_v() : \{d_{\text{puzzle}(v)}\} \\
&\forall p \in \text{mögliche Zielpositionen in } R_T : \\
&\quad \text{peek}_{u(R_T)}(\{p\}) : \{d'_p\} \\
&\quad \texttt{if } \text{passt}(d_{\text{puzzle}(v)}, d'_p) : \\
&\qquad \text{queue}_{u(R_C)}(\{d_{\text{puzzle}(v)}\}) \\
&\qquad \text{delete}_{u(R_S)}(\{d_{\text{puzzle}(v)}\}) \\
&\qquad \texttt{RESTART} \\
&\texttt{RESTART}
\end{aligned}
\tag{10.4}
$$

A_R sucht sich ein beliebiges Element aus R_S aus ($u(R_S)$ symbolisiert hier die Referenz von R_S) und bekommt dieses in Form einer Referenz v auf ein Puzzlestück, welches er anschließend liest. In allen verfügbaren Positionen bei R_T schaut der Agent die aktuellen dort erlaubten Randwerte für das Puzzlestück $d_{\text{puzzle}(v)}$ an. Die agenteninterne Funktion passt(x, d) berechnet, ob das konkrete Puzzlestück x mit den aktuellen Gegegebenheiten d zusammenpasst. Sollte es so sein, wird das Puzzlestück mitsamt der Positionsbeschreibung an die Kandidatenwarteschlange R_C übergeben und auf dem *Puzzle Piece Server* R_S gelöscht.

Ein Bastelagent ist in (10.5) beschrieben. Dieser arbeitet die Elemente ab (in Intervallen), die an die Warteschlange R_C übergeben worden sind. Es ist darauf zu achten, dass die Puzzlestücke in R_C geordnet sein müssen. Ein Agent kann nämlich lokales Wissen haben, wie oben angesprochen, dass Puzzlestücke angrenzend abgelegt werden können. Es kann aber stets davon ausgegangen werden, dass in R_C nur Puzzlestücke abgelegt worden sind, die zu einer früheren Zeit in R_T zulässig waren, also auch zu jedem späteren Zeitpunkt zulässig sind [2].

$$
\begin{aligned}
&\text{peek}_{u(R_C)}() : \{d_1, \ldots, d_n\} \\
&\forall d_i, 1 \leq i \leq n : \\
&\qquad \text{queue}_{u(R_T)}(\{d_i\}) \\
&\text{delete}_{u(R_C)}(\{d_1, \ldots, d_n\})
\end{aligned}
\tag{10.5}
$$

A_B liest jeweils eine geschlossene Menge von Puzzlestücken $d_1, \ldots, d_n$ mit ihren Positionen aus der Warteschlange R_C aus (insgesamt n Stück). Dann plaziert er die einzelnen d_is in der *Puzzletabelle* R_T anhand der angegebenen Positionen. Die

[2] Es wird angenommen, dass die Agenten die Puzzlestücke nicht verfälschen und nicht fälschen. Dazu später mehr im Teil 10.7.4.

Menge wird anschließend mit einem einzigen delete-Aufruf (ressourcenintern atomar) aus R_C entfernt. Da R_C strenge Monotonie auf der Ausgabeseite aufweist, muss für ein j mit $1 \leq j \leq n$ das Puzzlestück d_j entweder das erste noch zu entfernende Puzzlestück sein oder es sind alle n Puzzlestücke bereits durch einen anderen Agenten aus der Agentengruppe entfernt worden.

Die Monotonie der beiden Sequenzen ist hier offensichtlich, weil es nur eine einzige Schreiboperation gibt, die abschließend ausgeführt wird und die sich auf eine Eingaberessource bezieht.

Es ist darauf zu achten, dass das Puzzlebeispiel als Prozess nicht sehr robust ist. Es könnten Schwierigkeiten entstehen, wenn fehlerhafte Puzzlestücke in den Prozess kommen. Man sollte hier eventuell einplanen, dass solche fehlerhaften Puzzlestücke verworfen werden können. Eventuell würde sich auch eine zusätzliche Agentengruppe empfehlen, die den aktuellen Status von R_T untersucht und fehlende Puzzlestücke in R_S einreiht.

Für R_T wurde das Ordnungskriterium V gewählt, weil die Puzzlestücke strukturell zwar gleich sind, aber aus den obigen Robustheitsgründen mit Identifikationsnummern versehen sein sollten. Dieser Unterschied verhindert dann, beim Wiedereinreihen eines Puzzlestücks in die Ressource R_S, dass es aus Monotoniegründen herausgefiltert wird.

10.6 Verteilte Lösung eines Constraint-Problems

Die verteilte Lösung des n-Damen-Problems ist ein anderes Beispiel für einen kooperativen Prozess mit einem unterliegendem Constraint-Problem. Die Prozessteilnehmer kennen sich nicht und können miteinander nicht kommunizieren. Die Web-Anwendung dazu wurde mit Hilfe von *cgi2c* (siehe Kapitel 6) geschrieben und benutzt als Client den Web-Browser, der mit AJAX [Gar05] angetrieben wird.

Dazu wurde im Laufe des Experiments unsere Lehrstuhl-Webseite [LS112] mit einem Javascript-Programm [ECM02] [ECM09] präpariert, das asychrone Anfragen bei der Besichtigung unserer Webseiten durchgeführt hat, die der Web-Browser im Hintergrund unbemerkt ablaufen ließ. Der Ablauf war so gestaltet, dass eine Unteraufgabe, das noch nicht gelöst wurde von einer Unteraufgabenwarteschlange abgeholt worden ist. Daraufhin wurde die Laufzeit des Web-Clients gebraucht, um das zugewiesene Teilproblem zu lösen. Dann wurde die Lösung in Form einer Anfrage an eine Warteschlange mit Lösungen übergeben und von der Unteraufgabenwarteschlange entfernt.

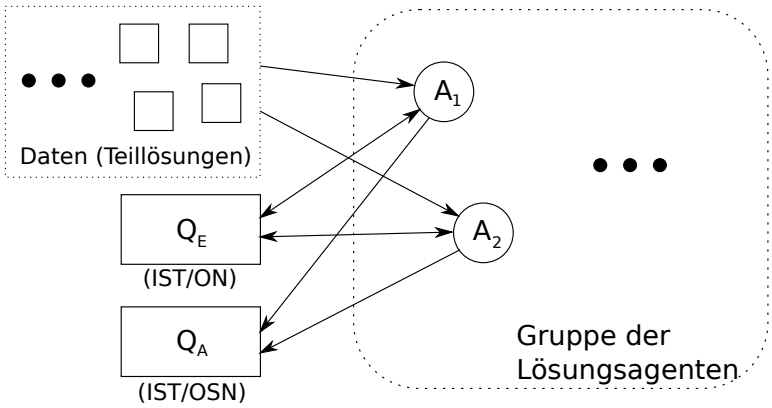

Abbildung 10.7: Systemaufbau zur Lösung des n-Damen-Problems

Die Abildung 10.7 veranschaulicht das System für den verteilten Prozess. Die Ressourcen auf der linken Seite sind alle auf dem Web-Server. Die Daten sind getrennt von der Kontrolle, die in Q_E und Q_A verwaltet wird. Eine beliebige Anzahl der Agenten in der Gruppe der Lösungsagenten für das n-Damen-Problem erhält das Agentenprogramm, welches in (10.6) spezifiziert ist.

$$
\begin{aligned}
&peek_{Q_E}() : \{y_1, \ldots, y_n\} \\
&\text{für ein fest gewähltes } i : u = \|y_i\|_{\text{uri}} \\
&\qquad \text{read}_u() : \{d\} \\
&\qquad \{e_1, \ldots, e_j\} = \texttt{löse}(d), \text{mit } j \in \mathbb{N}_0 \\
&\qquad \forall k \in 1, \ldots, j : \text{queue}_{Q_A}(e_k) \\
&\qquad \text{delete}_{Q_E}(\{y_i\}) \\
&\texttt{RESTART}
\end{aligned}
\tag{10.6}
$$

Der Agent liest von Q_E eine Untermenge von noch nicht gelösten Aufgaben. Er sucht sich eine davon mittels eines zufällig ausgewählten i aus. Er liest die konkrete Aufgabe mittels read_u aus, mit der *URI* u, die aus y_i mit der Interpretation $\|\cdot\|_{\text{uri}}$ extrahiert wurde. Der Agent löst intern die Teilaufgabe und liefert alle gefundenen Lösungen in Q_A ab. Anschließend wird die gewählte Aufgabe y_i aus Q_E gelöscht. Dann fängt der Agent mit der nächsten Teilaufgabe an.

Die Aufgaben wurden in Teilaufgaben aufgeteilt, indem auf dem Rechner ein Präfix der Lösung bis zu einer bestimmten Rekursionstiefe vorberechnet worden ist. Der Agent hat dann alle möglichen Suffixe, die in (10.6) als $e_1, \ldots, e_j$ gefunden worden sind einzeln an Q_A übergeben. Der Algorithmus stimmt auch, wenn es keine Lösungen für das gesamte Präfix gibt, also $j = 0$ gilt. Das abschließende delete_{Q_E} wurde ans Ende der Sequenz gestellt und erfüllt damit die Abschlussregel aus Teil 8.8.2.

10.7 Defizite und Vorschläge zur Optimierung

Zwar ist es vollkommen klar, dass der ressourcenorientierte Architekturstil nicht für jedes Problem effizient sein wird, aber man muss dennoch diese Unzulänglichkeiten kritisch zusammenfassen. Es ist genauso wichtig zu wissen wann ein Lösungsansatz adäquat erscheint, wie auch wann dieser schlecht ausfallen kann.

In den nächsten Teilen werden einige der größten Kritikpunkte benannt und Abhilfen gesucht, die passend zum Architekturstil sind, aber eines größeren Aufwands bedürfen, der in dieser Arbeit nicht mehr weiter erörtert worden ist.

10.7.1 Synchrone Arbeitsweise

Eines der markantesten Probleme des ressourcenbasierten Architekturstils ist die Abwesenheit eines synchronen Antriebs für das Prozessmanagement. Durch die Entkopplung von Systemen macht man diese voneinander unabhängig und steigert das Potenzial zur Verteilung und parallelisierter Arbeit. Auf der anderen Seite büßt man Effizienz ein, weil die lückenlose Abarbeitung in Form von synchron angesteuerten stark gekoppelten Methodenaufrufen hier vollständig fehlt.

Das Problem liegt darin, wie die Ausführung organisiert ist. Die datenorientierte Idee des ressourcenbasierten Architekturstils erlaubt es nicht, die Rechenkraft für das Prozessmanagement anzufordern.

Es ist natürlich uninteressant, diese Unzulänglichkeit anzugehen und gleichzeitig auf die klaren Vorteile der entkoppelten „echten“ Verteilung zu verzichten. Allerdings gibt es bereits Konzepte in der realen Welt, um mit Entkopplung umzugehen und trotzdem auf der Managementseite effizient zu sein.

Wie auch in der verteilt funktionierenden wirklichen Welt, gibt es Ereignisse, über die man nicht informiert wird. Es ist auch klar, dass man nicht jede Art von Änderung in der Welt öffentlich bekannt geben muss. Es würde vom Prinzip der Idee einer globalen Benachrichtigung entsprechen, welche synchronisierend wirkt. Deswegen hat man Mittel und Wege gefunden, in der verteilten Welt an Informationen zu kommen, indem man so etwas wie *Seitenkanäle* benutzt.

Diese Art von Kommunikationsverfahren lassen sich ebenfalls über den ressourcenbasierten Architekturstil legen, wie das in Abbildung 10.8 dargestellt ist.

Agenten können zwar miteinander nicht über die konkreten Inhalte kommunizieren, aber man könnte für synchrones Arbeiten einen Benachrichtigungsmechanismus etablieren. Dazu müsste natürlich eine entsprechende Infrastruktur angeboten werden. Denkbar wären Peer-To-Peer-Netze oder einfachere verteilte Subscription-basierte Verfahren. Dabei ist die Effizienz dieser Infrastruktur ausschlaggebend für die Latenzzeiten. Agenten müssten vom Konzept her erweitert

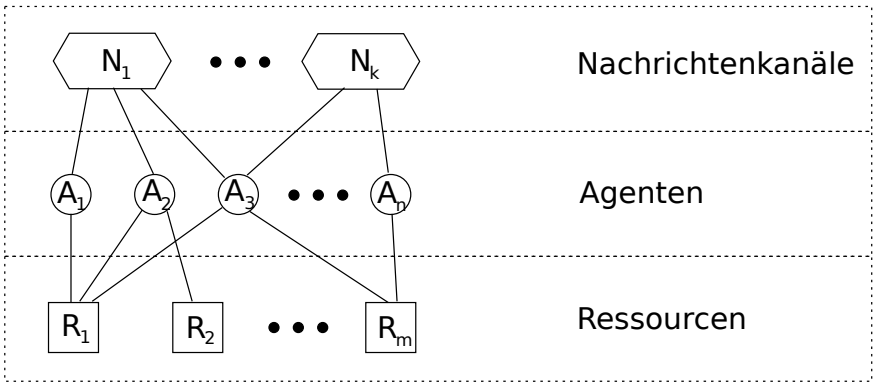

Abbildung 10.8: Vorschlag zur Umsetzung der synchronen Arbeitsweise

werden mit Mechanismen für die Beschreibung ihrer Interessen und den kommunikativen Möglichkeiten diesbezüglich. Auf diese Weise können Agenten sich gegenseitig über die Änderungen auf der Ressourcenmenge $\mathfrak{R}$ benachrichtigen, die sie interessieren.

Da Nachrichten in einem verteilten System verloren gehen können, sollte man darauf achten, dass sie keine Daten enthalten, die essentiell sind für die Fortführung des Prozesses. Das bedeutet, dass die Daten in Nachrichten ebenfalls wie die internen Agentenzustände bei Verlust den Prozess nicht stören dürfen.

10.7.2 Auslastung und Caching

Es ist sehr schwierig das Caching für die Ressourcen zufriedenstellend zu unterstützen, wie es im Teil 6.5 gezeigt wurde. Aber auch, wenn es gut funktioniert, ist das System in bestimmten Fällen unter erhöhter Last. Dies hängt damit zusammen, dass die *Pull-Strategie*, die im entkoppelten Architekturstil benutzt wird, die Agenten dazu bringt gehäuft Anfragen zu stellen, um festzustellen, ob sie sinnvolle Arbeit tun können.

Wegen dieser Arbeitsweise kommt es bei mancher Art von Prozessen gehäuft zum *Polling* [AS83] (aktives Warten), was nicht sehr schön ist. Hier kann die Lösung ebenfalls die synchronisierte Arbeitsweise sein, die im Teil 10.7.1 vorgestellt worden ist. Möchte man die Architektur aber so belassen wie sie ist und ohne solche Erweiterungen betreiben, dann empfiehlt es sich, die Gültigkeit der Ressourcenzustände für die Zukunft zu schätzen. Diese Strategie wird ebenfalls in *HTTP* verfolgt. Mit Hilfe von Metainformationen im Protokoll, lässt sich dort die voraussichtliche Gültigkeit für eine Ressourcenantwort festlegen.

Ist solche Einschätzung nicht möglich, kann man auf Polling nicht verzichten. Die einzige Möglichkeit, die einem noch bleibt, sind *konditionale Anfragen*, die

auch *HTTP* benutzt, um wenigstens die Bandbreite (auf dem jeweiligen Medium) zu schonen. Bei einer konditionalen Anfrage wird die Antwort nur in dem Fall geliefert, wenn sie sich gegenüber einem früher spezifizierten Zeitpunkt verändert hat. Ist dies nicht der Fall, dann gibt die Ressource eine besondere Antwort, die anzeigt, dass sich nichts verändert hat. Das Verfahren wurde im Teil 7.1.1 kurz erwähnt, aber müsste entsprechend modifiziert werden, damit der Agent keine exklusive essentielle Information hält, die in dem Architekturstil nicht konform ist. Die plausible Lösung ist, diese Information in Ressourcen abzuspeichern, um die Arbeit in Agentengruppen zu unterstützen.

10.7.3 Speicherkapazität

Wie schon im Teil 3.15 erwähnt, ist Speicher nicht sehr kritisch in Ressourcen. Andererseits wird der Speicher in vielen Fällen effektiv nicht mehr gebraucht, nachdem ihn jeder Agent verwertet hat, der diesen zu einer einmaligen Berechnung gebraucht hat. Die Speicherkapazität einzugrenzen, gehört hier vor allem zum Bereich der Optimierung.

Am Beispiel der Simulation der expliziten Synchronisierung von Aktivitäten im Teil 9.1.6 wurde ein Agent (dort A_T) verwendet, der Bestätigungen verarbeitet und eine Art Aufräumarbeit in Ressourcen durchgeführt hat. Das gleiche Prinzip lässt sich auch hier in vielen Fällen anwenden.

Einige monotone Warteschlangen brauchen allerdings alle Datensätze, um zu beurteilen, ob eine Information im System verarbeitet werden soll. Ist eine Information von der Aktualität viel älter, dann wird sie einfach ignoriert. Allerdings ist im schlimmsten Fall ein Agent sehr stark verzögert worden (man kann es theoretisch nicht begrenzen), sodass die Warteschlange eigentlich für immer für Monotonie am Eingang garantieren muss.

Was bleibt ist, anhand der Eingaberessourcen für Agenten abzuschätzen, ob es wahrscheinlich ist, dass ein Agent noch existiert, der eine dermaßen alte Information verarbeitet und diese zur Zerstörung der Monotonie in den Ausgabenressourcen führen könnte. Unterstützend dazu könnte man ein weiteres Kriterium zur Abbildung t (Ermittlung der Aktualität eines Elements oder Datums) hinzufügen, welches ein sehr altes Datum uneingeschränkt verwirft.

10.7.4 Trust und Sicherheit

Im Umfeld der Sicherheit können sich Inkonsistenzen in Systemen entwickeln, wenn sich Agenten nicht an vorgeschriebene Protokolle halten. Solche Arten von

Agenten wirken sich schädigend aus und können das Gesamtergebnis stören, dessen Unterprobleme behandelt werden.

Um dieses Problem anzugehen, muss man in die Ressourcenwelt den Begriff des Vertrauens einbringen. Dies muss man jedoch mit Vorsicht angehen, denn die ressourcenbasierte Architektur profitiert von der spontan entstehenden Rechenkraft, die anonyme Agenten anbieten. Es wäre also keine gute Idee, Ressourcen generell mit Authentifizierungsmechanismen zu versorgen, sodass Agenten individuell authorisiert werden können, Zustandsmodifikationen durchzuführen.

Das Ziel hierbei ist das Protokoll abzusichern. Vor allem muss man nachvollziehen können, ob der Agent tatsächlich Daten aus legitimen, bekannten Quellen verarbeitet hat. Um so eine Absicherung gegen Fälschungen zu realisieren, könnte man die Information kryptographisch signieren [W3C08] [PGP00].

Ein solches Verfahren, das man hier anwenden könnte, ist die Absicherung der Eingabedaten der Agenten mittels Signaturen, wenn die Daten aus vertrauenswürdigen Ressourcen angefordert werden. Sind Eingabedaten signiert und ist die Verarbeitung rechenzeitaufwändig, während die Verifikation des Ergebnisses es nicht ist, dann kann der Agent einen entsprechenden Nachweis erbringen, dass er mit legitimen Daten gerechnet hat. Voraussetzung ist natürlich, dass die Ressourcen gegenseitig ihre Signaturen kennen, diese überprüfen können und sich vertrauen. Eine Skizze, wie dieses Verfahren funktioniert ist die Abbildung 10.9.

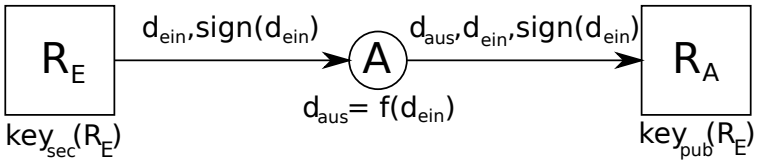

Abbildung 10.9: Illustration abgesicherter Eingabedaten

Die Eingaberessource R_E übergibt eine mit dem Geheimschlüssel $\text{key}_{\text{sec}}(R_E)$ signierte Eingabe d_{ein} an den Agenten A. Das Ergebnis der Signatur ist als $\text{sign}(d_{\text{ein}})$ übergeben worden. Der Agent A macht seine vorgesehene Berechnung $d_{\text{aus}} = f(d_{\text{ein}})$ und speichert diese in der Ressource R_A. Dabei wird die ursprüngliche Signatur für die Eingabe d_{ein} und d_{ein} mitübergeben. Da R_A den zugehörigen öffentlichen Schlüssel $\text{key}_{\text{pub}}(R_E)$ kennt, kann anhand der Eingabe und der zugehörigen Signatur die Echtheit verifizieren kann, kann in R_A schlussgefolgert werden, dass der Agent echte Eingabedaten d_{ein} aus R_E erfragt hat. Ist es zudem, wie oben beschrieben, einfach das Ergebnis d_{aus} aus der aufwändigen Berechnung mittels f zu verifizieren, wenn man die Eingabedaten hat, dann kann die gesamte Berechnung, die A durchgeführt hat, in R_A authorisiert werden.

Kapitel 11

Verwandte Arbeiten

In diesem Kapitel werden Arbeiten vorgestellt, die sich thematisch sehr nah mit den hier in der Arbeit aufgeführten Problemen befassen. Sie stellen einerseits alternative Sichten dar und andererseits waren sie eine große Inspiration, den agentengetriebenen ressourcenbasierten Architekturstil auszuarbeiten.

11.1 Architekturstile

Bevor andere relevante Arbeiten benannt werden, muss noch einmal daran erinnert werden, dass der größte Teil der Ideen, und überhaupt die Sicht auf die verteilten Architekturen, sich anhand der Dissertation von Roy Thomas Fielding [Fie00] orientiert und die größte Motivation für die Beschäftigung mit dem Thema gegeben hat.

Die datenbasierte Sicht auf Systeme, die nach dem Request-Reply-Prinzip angesprochen werden, erlaubt die Architekturen in zwei Schichten aufzutrennen, die völlig eigenständig betrachtet werden können: die Schicht der persistierenden Ressourcen und die Schicht der aktiven Agenten.

Der eingeführte Architekturstil, den viele inzwischen als *REST* kennen, ist mit den vielen guten Ideen auch praktisch realisiert worden. Wir kennen ihn als das *World Wide Web* und er ist unverzichtbar geworden, auch wenn einige der Ideen selbstständig geworden sind und sich in eine andere Richtung entwickelt haben, als Fielding sich das ursprünglich vorgestellt hat.

Bei dem hier behandelten ressourcenbasierten Architekturstil wird zunächst ein Schritt zurück zu den eigentlichen Vorstellungen, die aus der Dissertation stammen, gemacht. Die dort nicht weiter behandelten Themen, welche Möglichkeiten die Ressourcen anbieten und die gänzlich nicht besprochene Welt der Agenten ($\mathfrak{A}$) und der zugehörigen Protokolle (hier in der Form der Agentenprogramme eingeführt), stellt eine Fortentwicklung dieser Gedanken dar.

In der Arbeit von Ivan Zuzak, Ivan Budiselic und Goran Delac [ZBD11] wurde der Architekturstil REST berechtigterweise kritisiert. Es wurde behauptet, dass es keine formalen Ansätze gibt, den Nutzen von REST beschreiben zu können und das schon seit 10 Jahren. Es wurde konsequenterweise ein Modell vorgeschlagen, die Anwendungen als nichtdeterministische endliche Automaten auf der Grundlage der Hypermedia-Verknüpfungen zu beschreiben. Das Konzept, dass Ressourcen (dort „REST-Server“) leichtgewichtig sein sollen und dass die eigentliche Arbeit bei den Clients liegt, wurde ebenfalls erkannt. Die Spezifikation der Client-Seite mit Automaten ist natürlich auf die hier erbrachten Ergebnisse übertragbar und ist ein gültiger Ansatz, um Agentenprogramme zu beschreiben. Ein geeignetes Framework für die Programmierung von Agenten erleichtert die Anwendungsprogrammierung, aber schränkt gleichzeitig die Client-Vielfalt ein. Automatisierte Clients sind sicherlich ein mächtiges Werkzeug für Applikationen im ressourcenbasierten Architekturstil. Die Integration manueller Agenten sollte jedoch nicht unterschätzt werden. In diesem Sinne wurde innerhalb des ressourcenbasierten Architekturstils ein abstrakteres Modell für Agentenprogrammierung gewählt und sonst weitestgehend offen gelassen.

11.2 Konsistenz

Werner Vogels hat insbesondere für das World Wide Web festgestellt, dass Konsistenz bei der Zurverfügungstellung von Daten teilweise unter gewissen Bedingungen entkräftigt werden kann. Die „eventuelle Konsistenz“ wurde zunächst in einem seiner online veröffentlichen Artikel [Vog08] und dann im Journal *Communications of the ACM* [Vog09] untersucht. Diese Art von schwächerer Konsistenz kennt man bereits seit langem aus dem *Domain Name System* (DNS) [Moc87a] [Moc87b]. Während sich die Arbeit eher mit Datenspeicherung, Datenbanken und Informationssystemen befasst, sind die Ideen in den Abstraktionsmechanismus, den die Monotonie bietet, miteingeflossen. Die Ordnung, die bei der Ausbreitung von Informationen in ressourcenbasierten Architekturen, mittels der Kontrollressourcen realisiert wird, nimmt diese geschwächte Konsistenz in Kauf. Die größtmögliche Aktualität der Ressourcenzustände wird kooperativ erzeugt und sorgt deswegen verspätet für verteilte konsistente System- und Ressourcenzustände.

11.3 Ressourcen und ihre Eigenschaften

Ähnliche Arbeiten bezüglich verteilter Kommunikation wurden im Workshop *WS-REST 2010* der Konferenz WWW2010 vorgestellt. Das Papier mit dem Titel

A RESTful Messaging System for Asynchronous Distributed Processing [JR10] stellt eine Methode vor, wie man Daten in einem lose gekoppelten Netzwerk von Knoten propagieren kann. Es wurde dort ebenfalls erkannt, dass Idempotenz und Monotonie, die hier in der Arbeit verlangten Eigenschaften für Prozesssteuerung und Kopplung von Systemen haben und eine Architektur von bestimmten globalen Annahmen entlastet. Unter diesem Gesichtspunkt wurden dort verteilte Architekturen betrachtet, die unter anderem diese Eigenschaften in Form von Restriktionen als Grundlage haben.

Die resultierenden Architekturen sind insbesondere verteilte knotenbasierte Datenspeicher, die dynamisch Wissensmanagement betreiben, die sogenannten „Propagation Networks“, die in der Dissertation von Alexey Radul [Rad09] eingeführt worden sind und dazu erwähnenswert sind. Diese Arbeit verwirft ebenfalls die Idee der generischen Ressource, die lediglich einen Wert speichert und bezeichnet sie als unzureichend. Das entspricht dem Gedanken, dass Informationsorganisation mit Hilfe von Datenstrukturen, oder wie dort bezeichnet „Informationsakkumulation“ verwaltet werden sollte. Diese Arbeit beschäftigt sich allerdings vielmehr mit der verteilten Speicherung logischer Sachverhalte in Anwendungen für Informationssysteme und Typinferenz.

11.4 Idempotenz und Protokolle

Auf dem Gebiet der Netzwerkprotokolle wurde *Idempotenz* als Werkzeug bereits von Brown, Grossman und Knight [BGK02] erkannt, um vervielfachte Nachrichten in Netzwerken auszufiltern. Auch in diesem Fall der Kommunikation von jeweils zwei Endpunkten gibt es keine Möglichkeit, globale Zustände zu halten. Die Endpunkte sollen trotzdem zu einem Konsens kommen, ob eine Nachricht übermittelt wurde und zusichern, dass diese nur genau ein Mal verarbeitet wurde.

Obwohl in dem Papier die lokal vergebene eindeutige ID als Mittel eingesetzt wird (ähnlich dem Kriterium T bei den Warteschlangentypen), haben die Autoren nicht explizit die Eigenschaft als Monotonie benannt, deren Nutzen in dieser Arbeit über den ressourcenbasierten Architekturstil etwas gründlicher untersucht wurde.

Ein Netzwerkprotokoll ist zwar kein synchroner Request-Reply-Mechanismus, weist hier jedoch einige Gemeinsamkeiten auf. Dies liegt daran, dass das Verhalten mittels Zeitüberschreitungen gesteuert wird und implizit eine Zeitüberschreitung lokal als Fehler interpretiert. Auch die Netzwerkkomponenten werden hier, ähnlich wie die vorgestellten Warteschlangen, mit Puffern ausgestattet, um Nachrichten aus Effizienzgründen zu akkumulieren.

Obwohl sich hier viele Ideen überdecken, hat ein Netzwerk einen etwas anderen Charakter. Es liegt an den Agenten und dem Umgang mit der Kommunikation auf einer ressourcenbasierten Architektur. Agenten sind in der Lage mehrere Ressourcen zu adressieren, und dabei nicht nur Punkt-zu-Punkt-Übertragungen durchzuführen. Es wurde auch gezeigt, dass Agenten sich vielmehr dazu eignen, Zuständsübergänge in der Ressourcenwelt $\mathfrak{R}$ zu veranlassen und gleichzeitig, ihre Ausgaben konsistent für die angekoppelten Systeme bereitzuhalten.

Der ressourcenbasierte Architekturstil ist sicherlich ein allgemeinerer Formalismus als ein solches Kommunikationsnetzwerk, aber unterliegt auch weniger Einschränkungen, deswegen gibt es mehr Möglichkeiten und mehr Kommunikationsvielfalt.

11.5 Agenten und Kommunikation

Eine ganz interessante Form eines Architekturstils und eines Formalismus für verteilte Systeme sind die *Input-Output-Automaten* von Nancy A. Lynch [LT88] [Lyn96]. Der Charakter der optimistischen Form der Aufrufe auf den Komponenten („non-blocking“), das parallelisierte Verhalten beim Weiterreichen von Ausgaben zu Eingaben wurde im Modell der Agenten in dem hier eingeführten ressourcenbasierten Architekturstil verwertet.

Den I/O-Automaten fehlt allerdings die Sicht auf die datenbasierte Ressourcenwelt. Während die Komponenten innerhalb des I/O-Automaten-Modells direkt miteinander kommunizieren, sind solche direkten Kommunikationswege beim ressourcenbasierten Architekturstil nicht erlaubt. Es wurde hier explizit die Ressourcenwelt ($\mathfrak{R}$) eingeführt, damit der Ausführungszustand vollständig abgekoppelt wird von der Ausführungsebene der Agenten und damit ganz klar getrennt ist. Dies wurde aus zwei Gründen gemacht. Erstens ist ein Zustand einer Anwendung oder eines Prozesses im datenbasierten Charakter für Menschen besser verständlich. Und zweitens, lässt sich der datenbasierte Teil in der Praxis viel einfacher mit Redundanz ausstatten als die aktiven Teile einer Anwendung.

Während das I/O-Automaten-Modell sehr viele richtige Ansätze enthält und hinreichend mächtig ist, um beliebige Anwendungen zu betreiben, bringt es fast die vollständige Komplexität der Programmierung von verteilten Systemen und der verteilten Algorithmen mit auf die Programmierplattform.

Auffällig ist jedoch auch die Gemeinsamkeit der I/O-Automaten und des ressourcenbasierten Architekturstils. Beide Modelle verlassen sich auf die lokale Beschreibung des Systems innerhalb der Komponenten, die Aktionen ausführen können. Dies entspricht der *Choreographie*, die für den natürlichen Antriebsmechanismus bei verteilten Systemen unentbehrlich ist.

Weiterhin hat Charlton auf seinen Webseiten ebenfalls erkannt, dass ein agentenbasiertes Modell für REST-Applikationen geeignet ist [Cha10]. Er schlägt vor, einen REST-Agenten mit Sensoren und Effektoren auszustatten und die Ressourcen als seine Aktionsumgebung anzusehen. Die Vision, dass automatisierte Agenten nicht nur Web-Browser sind, die Web-Seiten anzeigen, sondern vielfältigere Tätigkeiten für den Benutzer erledigen können, wird auch bei seiner Beschreibung sehr deutlich. Weiterhin versteht Charlton den Antrieb eines Agenten ebenfalls als einen endlichen Automaten. Einer seiner Kommentatoren in dem Online-Artikel weist ihn darauf hin, dass der größte Teil der Logik in Agenten verlagert werden sollte und dass die Agenten reaktiv programmiert sein sollten. Ein anderer Kommentator weist Charlton auf RDF [RSS08] hin, welches nützlich ist um Ressourcen zu implementieren. Auch das ist ein Hinweis auf die Nutzung von strukturierten Ressourcen (datenstukturgetrieben), die weit mächtiger sind als generische Ressourcen. Diese Ideen hat Charlton bis heute noch nicht verarbeitet. Sie sind aber im Grunde wichtig für den ressourcenbasierten Architekturstil. Ferner wurde die weitere Analyse der Protokolle (Agentenverhalten) nicht betrachtet und die Mächtigkeit dieser Art von Architekturen ebenfalls nicht untersucht.

11.6 Workflows und Geschäftsprozessmodellierung

Eine frühe Idee der Programmierung von Workflows mit Hilfe von Modellen bietet die METAFrame-Umgebung [SMCB96]. Sie setzt die Idee um, „Programmierung im Großen“ zu betreiben, indem vorgefertigte wiederverwendbare Komponenten benutzt werden, um gröbere Abläufe zu modellieren. Nun ist das im Prinzip die Idee hinter Geschäftsprozessen und dem diensteorientierten Paradigma. Die Ideen aus METAFrame wurden weiter in diese Richtung entwickelt und sind die Grundlage für das jABC-Framework, welches von Ralf Nagel weiter ausgearbeitet wurde.

In Ralf Nagels Dissertation [Nag09] werden Geschäftsprozesse fachlich beschrieben, indem ihr Ablauf grafisch modelliert wird. Diese Arbeit folgt also primär dem Ansatz, Geschäftsprozesse mit Hilfe von formalen Modellen, die man auch als Workflows [Law97] [vdAtHKB03] kennt, zu beschreiben, die im Teil 1.2.1 und im Kapitel 3 vorgestellt worden sind.

Im weiterführenden Papier zum Thema SCA und jABC [JMN+08] wurde argumentiert, dass das jABC-Framework mit Diensteorientierung die Komplexität des Programmierens reduziert, weil in der Entwurfsphase vielmehr das Verhalten des Gesamtprozesses beschrieben werden kann, während der SCA-Ansatz

sich vielmehr auf Systemkomponenten konzentriert, was die Beschreibung des eigentlichen Verhaltens erschwere.

Ein wichtiger Einwand bezüglich der Orchestrierung von Prozessen wurde im Teil 1.3.4 erwähnt. Diese Form von Einfachheit schränkt dem Prozessdesigner die Sicht ein und gaukelt ihm eine sequentielle Arbeitsweise von Prozessen vor, wobei Leistungseinbußen bei natürlich verteilten Architekturen zu erwarten sind, wie zum Beispiel bei Systemen, die auf dem Web als Ausführungskontext implementiert worden sind. Hier in dieser Arbeit wurden Workflows an vielen Stellen als ein Beispiel für eine einfachere Form der Prozessausführung genommen. Deren zentralisierte Beschreibung (Modell) und die erforderliche zentralisierte Ausführungs- und Steuerungskomponente (Orchestrierung) behindern allerdings die Entfaltung der verteilten Ausführung, die vielen modernen Abläufen ganz natürlich zu Grunde liegt. Analog zu einer zentralen Ausführungseinheit für Prozesse lässt sich im ressourcenorientiertem Paradigma die Ausführung besser auf mehrere unabhängige Ausführungseinheiten verteilen. Für diese Aufgabe wurde explizit die Komponente des Agents ausgewählt, welche sich auch allgemein in verteilten Architekturen und Algorithmen findet.

Eines der wichtigsten Argumente hier ist also, die Ausführungskomponente nicht aus Einfachheitsgründen zu zentralisieren, weil sie leichter für Prozessentwickler zu behandeln ist. Die erhöhte Komplexität bei der natürlichen Parallelisierung sollte bei Anwendungen, die daraus Vorteile ziehen können unbedingt erhalten bleiben, um sie angemessen mit Entkopplung der Komponenten auszustatten. Diese entkoppelte Arbeitsweise der Anwendungsteile erlaubt bessere Verteilung der Rechenkraft bis hin zu massiv gestalteten Mehrbenutzersystemem, die sich für Szenarien wie dem Crowdsourcing eignen und nur schwierig als ein Prozess aufzufassen sind, weil die Rechenkraft unvorhergesehen zur Verfügung steht. Konsequenterweise lassen sich die Abläufe nicht planen und entstehen vielmehr spontan.

Es ist natürlich sinnvoll in dieser Arbeit Workflows und deren Paradigmen vorzustellen. Hiermit ist es möglich einen Vergleich mit den herkömmlichen Verfahren anzustellen. Auf diese Weise lassen sich die Stärken des ressourcenorientierten Architekturstils herauskristalisieren.

Der praktische Nutzen der Geschäftsprozesse, jABC und die daraus resultierenden Einsichten [MSR06] gehörte in dieser Arbeit zum Hintergrundwissen das sehr wichtig war, um die Qualität der hier angestrebten Lösungen beurteilen zu können. Im Teil 9.1 wurden deswegen die Konzepte, die aus den Workflows stammen simuliert, um zu zeigen, dass ressourcenbasierte Architekturen diese Konzepte grundlegend unterstützen können. Diese Eigenschaften und die direkten Assoziationen zwischen der Welt der Prozesse und des verteilten World

Wide Web sind aus der Arbeit von Roy Thomas Fielding [Fie00], die den Architekturstil REST in die Welt der Geschäftsprozesse eingebracht hat, nicht zu entnehmen.

Die theoretische Grundlage für Workflows, die Petri-Netze [Pet62], sind im Teil 9.2 behandelt worden. Wie vermutet, stellte es sich als schwierig heraus, eine Äquivalenz zu den Petri-Netzen nachzuweisen, wegen des Umgangs mit Nichtdeterminismus bei der Wahl der Transitionen. Dieses Problem der Implementierung einer Ausführungsumgebung ist bekannt [JK09] und kann einigermaßen gut durch den Nichtdeterminismus gelöst werden, der sich bei der Wahl des Agenten zum Schalten einer Transition ergibt.

In der Welt der Workflows steht das Modell, also der ausformulierte, vorbereitete Ablauf im Zentrum. Die ressourcenbasierten Architekturen hingegen lassen den endgültigen Ablauf zunächst offen und überlassen die Wahl den Agenten, welchen Weg die Ausführung nimmt anhand der gerade zur Verfügung stehenden Alternativen für die Fortführung. Damit ist der Architekturstil grundsätzlich parallel gestaltet, im Gegensatz zu den Workflows, in denen die Ausführungssequenz bei den darunterliegenden Modellen dominierend ist. Dieses Problem und die Folgen davon wurden im Teil 3.2 näher erörtert.

11.7 Geschäftsprozesse und Artefakte

Artefakte (oder in Englisch „business artifacts“) stellen einen neuen Ansatz dar für den Antrieb von Prozessen, die vor allem datenbasierten Charakter haben [CH09]. Die Idee dahinter ist, dass Workflows als Modell für einige Arten von Geschäftsprozessen einfach ungeeignet sind. Sie stellen einen Kompromiss dar, um mit Geschäftsobjekten flexibler umgehen zu können und sie trotzdem in die Business Process Management Umgebungen stellen zu können.

Das Prinzip welches sich dahinter verbirgt ist, die Geschäftsobjekte, die für Prozesse nötig sind, strukturiert mit Informationen anzureichern, die sich auf die Prozesskontrolle auswirken können. Oft denkt man dabei an Dokumente, die einen Bearbeitungszustand haben, der an die Instanzen der Geschäftsobjekte „angeheftet“ wird. Geht man zum Beispiel von einer Rechnung als Geschäftsobjekt aus, kann sie ein Kontrollfeld „bezahlt“ bekommen, sodass ein Prozess oder direkt das Prozessmanagement entscheiden kann, was mit einer konkreten Instanz einer solchen Rechnung zu tun wäre. Diese Kontrollinformation wird in der Terminologie der Artefakte „Attribut“ genannt und kann für deklarativ definierte Workflows gebraucht werden (zum Beispiel auf der Basis von Constraints und Pre-/Post-Conditions).

Die Steuerung geht hier also von den Daten aus, was ein wünschenswertes Ziel ist, welches im ressourcenbasierten Modell ebenfalls verfolgt worden ist. Ferner wurde aus den Geschäftsobjekten (die ihrem Charakter nach eher Daten sind) die Kontrollinformation für den Ablauf eines Prozesses extrahiert. Die Trennung von Kontrolle und Daten wurde hier ebenfalls als ein wichtiges Mittel erkannt.

Es ist wichtig festzustellen, dass die Kontrolle in ressourcenbasierten Architekturen vollkommen anders funktioniert, aber andererseits die Idee der Artefakte durchaus unterstützt werden kann. Während in der traditionellen Welt der Geschäftsprozesse man dokumentenorientiert denkt und die Artefakte ebenfalls durch Bearbeitungsschritte durchgereicht werden, ist der ressourcenbasierte Ansatz so konzipiert, dass die Rechenkraft zu den gespeicherten Daten zur Verfügung gestellt wird. Während ein Artefakt ein festgelegtes Ziel hat, wo es als nächstes verarbeitet wird, bleiben Ressourcen an ihrem Ort, sodass sie bereit sind, Daten an jeden interessierten Agenten zu liefern, der auf sie zugreifen darf. Artefakte können sehr gut mit Hilfe der *Indirektion* (siehe Teil 8.1) unterstützt werden. Dies realisiert man mit einer weiteren Ressource, die zum Beispiel alle „bezahlten Rechnungen“ als Referenzen speichert. Die Kontrolle wird hierbei vollständig auf eine andere Ressource ausgelagert und stellt nicht nur ein Attribut dar. Das Geschäftsobjekt ist sozusagen vollständig kompatibel mit älteren Prozessen, die keine Unterstützung für die Bezahlung von Rechnungen implementiert haben. Prozesse können hier auf diese Weise vollständig entkoppelt arbeiten.

Ressourcenbasierte Architekturen erlauben die Integration von Prozessen, die etwas langlebiger ist. Da die datenbasierten Informationen im Allgemeinen wegen des *Cool URI* Prinzips nicht verloren gehen, können Prozesse an Ressourcen angekoppelt werden, deren Daten schon längst durch andere Prozesse verarbeitet worden sind. Die Daten werden auch nicht durch alte Zustandsinformationen aus alten Prozessen beeinflusst. Man schafft dazu neue Ressourcen zur Zustandssteuerung und lässt Agenten mit neueren Agentenprogrammen die Arbeit machen (oder steuert gar Menschen, diese Arbeit zu übernehmen).

Der Ansatz der Business Artifacts greift nicht das Thema der massiven Parallelisierung und der Verwaltung redundanter Rechenkraft auf, wie sie für kooperative Prozesse benötigt wird.

11.8 Kollaborative Systeme

Ein *kollaboratives System* wie zum Beispiel ein *Blackboard* [Cor03] erlaubt verteilte Mitarbeit an einem verteilten Problem. Der Ansatz stammt aus der Welt der

künstlichen Intelligenz, wo an größeren Probleme mit Hilfe von kooperierenden Software-Komponenten gearbeitet wurde.

Im Verhältnis zu ressourcenbasierten Datenstrukturen ist ein Blackboard eine Synchronisierungkomponente für einen kooperativen Prozess. Entsprechend der Idee ist es mit einer internen Logik ausgestattet, um Teilergebnisse zu einer Gesamtlösung zu integrieren und zusammenzufassen. Ähnlich dem Blackboard, erlaubt eine Warteschlange mit wahlfreier Ausgabe, Nichtdeterminismus bei der Auswahl der Teilaufgaben zu benutzen, um die Präferenz zu Gunsten der parallelisierten Abarbeitung zu verschieben.

Ein gewichtiger Unterschied ist, dass ein *Blackboard* gleichzeitig zur Analyse und als Eingabe für Agenten dient, um die jeweils nächsten Teilprobleme zu bestimmen. Die Logik der Bildung von Abhängigkeiten zwischen den zu berechnenden Teilaufgaben ist innerhalb des Blackboards verborgen und bedarf sorgfältiger Überlegungen bezüglich Optimierungen [CGJ87]. Dies muss in ressourcenbasierten Systemen nicht der Fall sein. Die Generierung oder das Auffinden von Teilaufgaben in einer einzelnen Komponente, die nicht schwergewichtig im ressourcenbasierten Architekturstil sein sollte, ist zwar möglich, aber ungünstig für größere, rechenaufwändigere Szenarien. Deswegen ist es üblich, die zwei Phasen der Generierung von Teilaufgaben und dem Einsammeln der Teilergebnisse auf zwei (Sub)Systeme auszulagern. Auf diese Weise wird entkoppelte Arbeit in den beiden Phasen möglich, die sich, rein vom Verständnis her, überlagern dürfen. Es ist sogar der Fall, dass die Validitätsuntersuchung und die Integration der Teilergebnisse in ein weiteres (Sub)System ausgelagert werden kann. Im Falle von Blackboards werden hingegen sehr komplexe Synchronisierungsmechanismen eingesetzt [RRY10], um das System zu verteilen.

Die Paradigmen unterscheiden sich hier also. In einem ressourcenbasierten Architekturstil verzichtet man auf Berechnungen und Analysen in den Verwaltungs- und Kontrollressourcen und geht mit diesen *datenbasiert* um. Die Anwendungslogik verbirgt sich im stärkeren Maße bei den Agenten, die in dieser Art von Architekturen die rechenstärkste Komponente darstellen.

Kapitel 12

Schlusswort

»Unterschätze nie einen Menschen der einen Schritt zurück macht. Er könnte Anlauf nehmen!«

Urheber unbekannt

Das letzte Jahrzehnt in der Software-Entwicklung bringt sicherlich viele Vereinfachungen in weit verbreiteten und akzeptierten Konzepten. Im Falle der Geschäftsprozesse wurden Modelle entwickelt, die sehr plausibel und explizit anhand eines globalen Steuerungsmechanismus, ein Unternehmen steuern können. Diese Konzepte sind ausreichend und erfüllen den Zweck für eine ganze Reihe von Anforderungen, die für ein abgeschlossenes Unternehmen üblich sind und dessen Betrieb in festen Organisationsstrukturen und Abläufen verwalten.

Mit der großartigen Architektur des World Wide Web, gibt es ganz andere Organisationsmöglichkeiten. Die Abläufe, die vom Charakter auf paralleles und entkoppeltes Arbeiten ausgelegt sind, lassen sich nicht sehr gut in eine feste und starre Form bringen. Die vor allem datenbasierte Architektur bietet in erster Linie keinen Ansatzpunkt, koordiniert und in systemübergreifender Kooperation solche Abläufe, die vor allem auf Verteilung ausgelegt sind, gut zu unterstützen.

Die erste vernünftige Reaktion sollte sein, sich anzuschauen woran es liegt, dass das WWW bei dem großen Potenzial nur so wenig bekannte Mittel für die Steuerung bereit hält. Die Analyse der Modelle in dieser Arbeit zeigte, dass die zentralisierte Orchestrierung sich auf eine echt verteilte Architektur nur mit dem Inkaufnehmen einiger Nachteile übertragen lässt. Die Entscheidung hier war, zu schauen, wie sich verteilte Systeme früher gestaltet haben, bevor damit angefangen wurde, die natürliche Verteilung bezüglich Kontrolle in diesen Systemen

dermaßen ungünstig einzuschränken. Dieser Schritt zurück war zuerst nötig. Die Erkenntnisse und vor allem die Vorteile aus herkömmlichen Modellen wurden ebenfalls evaluiert und in die Entscheidungen miteingeschlossen.

Es ist im Allgemeinen sehr schwierig, Konzepte im Fachgebiet der Informatik einzuführen, die generell sind und allgemeingültig. Es wäre auch illusorisch, einen Architekturstil zu suchen, der alle Probleme dieser Themenbereiche angreift. Der ressourcenbasierte Architekturstil eignet sich hervorragend zur Unterstützung von Architekturen, die vor allem datenbasiert operieren und deren natürlicher Charakter durch parallele und entkoppelte Abläufe dominiert wird. Prozesse, die zu solchen Architekturen passen, sind die kooperativen Prozesse. Hier arbeiten mehrere unabhängige Applikationen und unterstützen sich gegenseitig beim Erreichen ihrer individueller Ziele. Dabei wird die Parallelisierung, falls nötig, aufgelöst, um Synchronisierung zu erreichen. Auf diese Weise lässt sich ein anwendungsübergreifender Konsens über ein Datum erreichen, welches gemeinsam gebraucht wird.

Nach der Einführung des Architekturstils und seiner innewohnenden Eigenschaften wurden anhand von Experimenten zahlreiche Überlegungen gemacht. Einerseits ging es darum, die Ressourcen mit Logik anzureichern, ohne das Request-Reply-Paradigma zu beeinflussen und dual dazu wurden in der Welt der Agenten prinzipielle Regeln aufgestellt, um Information für Anwendungen und Systeme konsistent bereitzustellen. Dazu wurde einerseits die Speicherung in Form von Ressourcenzuständen und andererseits die Ausbreitung der Information in ressourcenbasierten Systemen untersucht. Die hier erbrachten Ergebnisse wurden stets unter dem Gesichtspunkt gemacht, dass sowohl Entkopplung als auch Verteilung entsprechend gewahrt bleiben, ohne zurück zu fallen in alte und viel zu generelle Denkschemata, die zu den unerwünschten Nebeneffekten führen würden.

Viele Überlegungen dieser Arbeit bezogen sich darauf, zu zeigen, dass ein ursprünglich datenbasiertes Modell eine Mächtigkeit erlangen kann, die äquivalent zu den herkömmlichen, allseits bekannten Workflow-Modellen ist, die bevorzugt bei Geschäftsprozessen zum Einsatz kommen. Die Begründung basierte darauf, dass die einzelnen Merkmale im Workflow-Management simuliert werden können. Auch das theoretische Modell der Petrinetze wurde simuliert, wenn auch hier einige spezielle Wege gegangen worden sind, um die Zentralisierung zu vermeiden. Das Management von Petrinetzen stellte sich durch ihren nichtdeterministischen Charakter beim Schalten, praktisch als sehr schwierig heraus. Trotzdem wurde ein Weg gefunden, um das Schalten dezentral und lokal zu verwalten indem ein Kompromiss gefunden worden ist. Mit Hilfe der Konfliktmengenkonstruktion wurde die Gültigkeit des Schaltvorgangs eingeschränkt und stellte damit ein Mittel zur Dezentralisierung dar.

Die wahre Stärke zeigen ressourcenbasierte Architekturen jedoch in der Besonderheit der kooperativen Prozesse. Sie lassen Anwendungen und Systeme entkoppelt arbeiten und erbringen Ergebnisse in gemeinsamer Kooperation. Es wurden sowohl Einzelanwendungen demonstriert, die in kleiner Dimension Informationsflüsse zwischen entkoppelten Systemen realisieren, als auch größere verteilte Szenarien, die mit massiver Verteilung ausgewählte Constraint-Probleme angegriffen haben. Es werden zur Zeit zahlreiche Anwendungen im Bereich des Crowdsourcing entwickelt, die demonstrieren, wie man Menschen massiv in die Lösung von Problemen mittels Kooperation miteinbeziehen kann. Diese Öffnung geschäftsinterner Problematiken gegenüber der Öffentlichkeit hat ein weitreichendes Potenzial und ist viel flexibler im Gegensatz zum Outsourcing, wo ein Problem und seine Lösung auf eine bestimmte dritte Partei ausgelagert wird.

Der Ursprungsgedanke, so wenig Kontrolle wie möglich und genau so viel wie nötig zu exerzieren, hat einen Architekturstil entstehen lassen, der sich gleichgültig verhält bezüglich der Arbeitskraft, die gerade innerhalb der jeweiligen Anwendung zur Verfügung steht. Ferner sind die Anwendungen, ähnlich wie in der Welt der funkionalen Programmierung, nicht primär durch das „Wie“ beschrieben. Das „Wie“ ist vielmehr auf der Seite der Agenten zu beschreiben und bietet dort das Potenzial zur Optimierung. Die Ressourcenseite hingegen definiert lediglich alles Nötige für die Restriktionen der Verhaltensweise der Agenten, sodass das Ziel klar verständlich ist. Ergänzt wird die Funktionsweise in diesem Architekturstil durch Vereinbarungen bezüglich Protokollen auf der Basis von Agentenprogrammen.

Alles in allem bietet der ressourcenbasierte Architekturstil, das Potenzial für eine alternative Methode der Software-Entwicklung im Bereich der entkoppelten und verteilten Umgebungen. Durch seine Offenheit und den datenbasierten Charakter werden Systeme besser verständlich, lassen sich nach und nach zunehmend mit Automatisierung ergänzen und erlauben trotzdem den menschlichen Eingriff als letzten Ausweg, bevor ein System still steht. Wobei, wie es bereits mehrfach gesagt und als eines der Ziele festgelegt wurde, der Stillstand in diesen Architekturen, also die Abwesenheit von Rechenkraft, bedeutet nicht einen generellen Defekt am System.

Literaturverzeichnis

[AFM+05] AKKIRAJU, RAMA, JOEL FARRELL, JOHN MILLER, MEENAKSHI NAGARAJAN, MARC-THOMAS SCHMIDT, AMIT SHETH und KUNAL VERMA: *Web Service Semantics - WSDL-S*, 2005. http://www.w3.org/Submission/WSDL-S/.

[AHV85] ANDRE, F., D. HERMAN und J. P. VERJUS: *Synchronisation of Parallel Programs*. North Oxford Academic, 1985.

[AMG+95] ALONSO, G., C. MOHAN, R. GUNTHOR, D. AGRAWAL et al.: *Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management*. Proceedings IFIP, Working Conference on Information Systems for Decentralized Organizations, 1995.

[Apa12a] APACHE FOUNDATION: *Apache HTTP Server Project*, 2012. http://httpd.apache.org/.

[Apa12b] APACHE FOUNDATION: *Apache Tomcat*, 2012. http://tomcat.apache.org/.

[AS83] ANDREWS, GREGORY R. und FRED B. SCHNEIDER: *Concepts and Notations for Concurrent Programming*. ACM Computing Surveys, 1983.

[Bel09] BELL, DAN: *The Crowdsourcing Handbook - The How to on Crowdsourcing, Complete Expert's Hints and Tips Guide by the Leading Experts, Everything You Need to Know about*. Emereo Pty Ltd, 2009.

[BG81] BERNSTEIN, PHILIP A. und NATHAN GOODMAN: *Concurrency Control in Distributed Database Systems*. ACM Computing Surveys, 1981.

[BGK02] BROWN, JEREMY, J. P. GROSSMAN und TOM KNIGHT: *A Lightweight Idempotent Messaging Protocol for Faulty Networks*. SPAA, 2002.

[BL90] BERNERS-LEE, TIM: *Information Management: A Proposal*, 1990. http://www.w3.org/History/1989/proposal.html.

[BLFIM09] BERNERS-LEE, T., R. FIELDING, U. C. IRVINE und L. MASINTER: *Uniform Resource Identifiers (URI): Generic Syntax*, 2009. http://www.ietf.org/rfc/rfc2396.txt.

[Bre00] BREWER, ERIC A.: *Towards robust distributed systems*, 2000. Invited Talk.

[Cas81] CASANOVA, MARCO ANTONIO: *The Concurrency Control Problem for Database Systems (LNCS 116)*. Springer-Verlag, 1981.

[CB74] CHAMBERLIN, DONALD D. und RAYMOND F. BOYCE: *SEQUEL: A Structured English Query Language*. SIGFIDET '74 Proceedings, 1974.

[CGJ87] CORKILL, DANIEL D., KEVIN Q. GALLAGHER und PHILIP M. JOHNSON: *Achieving Flexibility, Efficiency, and Generality in Blackboard Architectures*. AAAI 87 Proceedings, 1987.

[CH09] COHN, DAVID und RICHARD HULL: *Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes*. IEEE, 2009.

[Cha10] CHARLTON, S.: *Building a RESTful Hypermedia Agent - Part 1*, 2010. http://www.stucharlton.com/blog/archives/2010/03/building-a-restful-hypermedia.html.

[Cor03] CORKILL, DANIEL D.: *Collaborating Software - Blackboard and Multi-Agent Systems and the Future*. Proceedings of the International Lisp Conference, 2003.

[CPP05] *ISO/IEC 9899:1999: C Programming Language Specification*, 2005. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[dh05] HÓRA, BILL DE: *HTTPLR - An Application Protocol for Reliable Transmission of Messages Using HTTP*, 2005. http://www.dehora.net/doc/httplr/draft-httplr-01.html.

[Dij71] DIJKSTRA, E. W.: *Hierarchical Ordering of Sequential Processes*. Acta Informatica 1, 1971.

[Dij74] DIJKSTRA, E. W.: *Self-stabilizing Systems in Spite of Distributed Control*. Communications of the ACM, 1974.

[DK75] DEREMER, FRANK und HANS KRON: *Programming-in-the large versus programming-in-the-small*. SIGPLAN Not., 1975.

[Eag85] EAGER, DEREK L.: *A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing*. SIGMETRICS '85 Proceedings, 1985.

[ECM02] *Information technology - ECMAScript language specification*, 2002. `http://standards.iso.org/ittf/PubliclyAvailableStandards/c033835_ISO_IEC_16262_2002(E).zip`.

[ECM09] *ECMAScript Language Overview*, 2009. `http://www.ecmascript.org/es4/spec/overview.pdf`.

[Edl12] EDLICH, PROF. DR. STEFAN: *Your Ultimate Guide to the Non - Relational Universe!*, 2012. `http://nosql-database.org/`.

[EGL+76] ESWAREAN, K.P., J.N. GRAY, R.A. LORIE, I.L. TRAIGER et al.: *The Notions of Consistency and Predicate Locks in a Database System*. Communications of ACM, 1976.

[Erl07] ERL, THOMAS: *SOA Principles of Service Design*. Prentice Hall, 2007.

[F+99] FIELDING, ROY THOMAS et al.: *Hypertext Transfer Protocol – HTTP/1.1*, 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[FB96] FREED, N. und N. BORENSTEIN: *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, 1996. `http://www.ietf.org/rfc/rfc2046.txt`.

[Fie00] FIELDING, ROY THOMAS: *Architectural Styles and the Design of Network-based Software Architectures*. Doktorarbeit, University of California, Irvine, 2000.

[Fie08] FIELDING, ROY THOMAS: *REST APIs must be hypertext-driven*, 2008. `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`.

[Gar05] GARRETT, JESSE JAMES: *Ajax: A New Approach to Web Applications*, 2005. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`.

[Ger09] GERHARDS, R.: *The Syslog Protocol*, 2009. `http://www.ietf.org/rfc/rfc5424.txt`.

[GHM+07] GUDGIN, MARTIN, MARC HADLEY, NOAH MENDELSOHN, JEAN-JACQUES MOREAU, HENRIK FRYSTYK NIELSEN, ANISH KARMARKAR und YVES LAFON: *SOAP Version 1.2 Part 1: Messaging*

Framework (Second Edition), 2007. http://www.w3.org/TR/soap12-part1/.

[GL02] GILBERT, SETH und NANCY LYNCH: *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*. ACM SIGACT News, vol. 33, issue 2, 2002.

[GoL12] *The Go programming language*, 2012. http://golang.org/.

[Goo11] GOOGLE: *Google Maps API Family*, 2011. http://code.google.com/apis/maps/.

[GR92] GRAY, JIM und ANDREAS REUTER: *Transaction Processing. Concepts and Techniques*. Morgan Kaufmann, 1992.

[GR04] GREINER, ULRIKE und ERHARD RAHM: *Quality-Oriented Handling of Exceptions in Web-Service-Based Cooperative Processes*. Proceedings of EAI-Workshop 2004, Enterprise Application Integration, 2004.

[Gra81] GRAY, JIM: *The Transaction Concept: Virtues and Limitations*, 1981.

[Gri11] *Grizzly Servlet Container*, 2011. http://java.net/projects/grizzly-servlet-container.

[Hoa78] HOARE, CHARLES ANTONY RICHARD: *Communicating Sequential Processes*. Communications of the ACM, 1978.

[Hoa04] HOARE, CHARLES ANTONY RICHARD: *Communicating Sequential Processes*. Prentice Hall, 2004.

[HR83] HAERDER, THEO und ANDREAS REUTER: *Principles of transaction-oriented database recovery*. ACM Computing Surveys (CSUR), 1983.

[HRW09] HUBERMAN, BERNARDO A., DANIEL M. ROMERO und FANG WU: *Crowdsourcing, attention and productivity*. Journal of Information Science, 2009.

[HW03] HOHPE, GREGOR und BOBBY WOOLF: *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[IET12] *The Internet Engineering Task Force (IETF)*, 2012. http://www.ietf.org/.

[Jav12] *Lernen über Java-Technologie*, 2012. http://www.java.com/de/about/.

[Jer12] *Jersey - JAX-RS (JSR 311) Reference Implementation for building RESTful Web Services*, 2012. http://jersey.java.net/.

[JK09] JENSEN, KURT und LARS M. KRISTENSEN: *Coloured Petri Nets, Modelling and Validation of Concurrent Systems*. Springer-Verlag, 2009.

[JMN+08] JUNG, GEORG, TIZIANA MARGARIA, RALF NAGEL, WOLFGANG SCHUBERT, BERNHARD STEFFEN und HORST VOIGT: *SCA and jABC: Bringing a Service-Oriented Paradigm to Web-Service Construction*. ISoLA 2008, CCIS 17, 2008.

[JR10] JACOBI, IAN und ALEXEY RADUL: *A RESTful Messaging System for Asynchronous Distributed Processing*. WS-REST 2010, preliminary proceedings, 2010.

[JSO12] *Introducing JSON*, 2012. `http://www.json.org/`.

[KJA93] KELLER, ARTHUR M., RICHARD JENSEN und SHAILESH AGARWAL: *Persistence software: bridging object-oriented programming and relational databases*. SIGMOD'93 Proceedings, 1993.

[KN11] KLEIN, URI und KEDAR S. NAMJOSHI: *Formalization and Automated Verification of RESTful Behavior*. Technical Report, 2011.

[Kou03] KOUBARAKIS, M.: *Multi-Agent Systems and Peer-to-Peer Computing: Methods, Systems and Challanges*. International Workshop on Cooperative Information Agents (CIA 2003), 2003.

[Lam78a] LAMPORT, LESLIE: *Implementation of Reliable Distributed Multiprocess Systems*. Computer Networks 2, 1978.

[Lam78b] LAMPORT, LESLIE: *Time, Clocks, and the Ordering of Events in a Distributed System*. ACM, 1978.

[Law97] LAWRENCE, P.: *Workflow Handbook 1997*. John Wiley and Sons, 1997.

[LBK01] LEWIS, PHILIP M., ARTHUR BERNSTEIN und MICHAEL KIFER: *Database and Transaction Processing*. Addison Wesley, 2001.

[LCR10] LIHOREAU, MATHIEU, LARS CHITTKA und NIGEL E. RAINE: *Travel optimization by foraging bumblebees through readjustments of traplines after discovery of new feeding locations*. The American Naturalist, 2010.

[LF81] LYNCH, N. und M. J. FISCHER: *On Describing the Behavior and Implementation of Distributed Systems*. Theoretical Computer Science vol. 13, 1981.

[Lip75] LIPTON, RICHARD J.: *Reduction: a method of proving properties of parallel programs*. Commun. ACM, 1975.

[LS112] *LS14 Webseite*, 2012. http://ls14-www.cs.tu-dortmund.de/.

[LT88] LYNCH, NANCY A. und MARK R. TUTTLE: *An Introduction to Input/Output Automata*. Technical Memo TM.373, 1988.

[Luc05] LUCKHAM, DAVID: *The Power of Events - An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Pearson Education, 2005.

[Lyn96] LYNCH, NANCY A.: *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc, 1996.

[Mar83] MARTIN, JAMES: *The Data-Base Environment*. Prentice-Hall, 1983.

[MBH+04] MARTIN, DAVID, MARK BURSTEIN, JERRY HOBBS, ORA LASSILA, DREW MCDERMOTT, SHEILA MCILRAITH, SRINI NARAYANAN, MASSIMO PAOLUCCI, BIJAN PARSIA, TERRY PAYNE, EVREN SIRIN, NAVEEN SRINIVASAN und KATIA SYCARA: *OWL-S: Semantic Markup for Web Services*, 2004. http://www.w3.org/Submission/OWL-S/.

[Mec12] *Amazon Mechanical Turk - Artificial Artificial Intelligence*, 2012. https://www.mturk.com/.

[Moc87a] MOCKAPETRIS, P.: *Domain Names - Concept and Facilities*, 1987. http://www.ietf.org/rfc/rfc1034.txt.

[Moc87b] MOCKAPETRIS, P.: *Domain Names - Implementation and Specification*, 1987. http://www.ietf.org/rfc/rfc1035.txt.

[MRMK09] MARINOS, ALEXANDROS, AMIR RAZAVI, SOTIRIS MOSCHOYIANNIS und PAUL KRAUSE: *RETRO: A Consistent and Recoverable RESTful Transaction Model*. International Conference on Web Services, 2009.

[MS96] MARGARIA, TIZIANA und BERNHARD STEFFEN: *Automatic Synthesis of Design Plans in METAFrame*, 1996.

[MSR06] MARGARIA, TIZIANA, BERNHARD STEFFEN und MANFRED REITENSPIESS: *Service-Oriented Design: The jABC Approach*. Dagstuhl Seminar Proceedings, Service Oriented Computing (SOC), 2006.

[Mur89] MURATA, TADAO: *Petri Nets: Properties, Analysis and Applications*. Proceedings of the IEEE, 1989.

[Nag09] NAGEL, RALF: *Technische Herausforderungen modellgetriebener Beherrschung von Prozesslebenszyklen aus der Fachperspektive: Von der Anforderungsanalyse zur Realisierung*. Doktorarbeit, Technische Universität Dortmund, 2009.

[NS05] NOTTINGHAM, M. und R. SAYRE: *The Atom Syndication Format*, 2005. `http://www.ietf.org/rfc/rfc4287`.

[OAS07] OASIS WEB SERVICES BUSINESS PROCESS EXECUTION LANGUAGE (WSBPEL) TECHNICAL COMMITTEE: *OASIS Web Services Business Process Execution Language*, 2007. `http://www.oasis-open.org/committees/wsbpel/`.

[OAS12] OASIS WS-BPEL EXTENSION FOR PEOPLE (BPEL4PEOPLE) TECHNICAL COMMITTEE: *OASIS WS-BPEL Extension for People*, 2012. `http://www.oasis-open.org/committees/bpel4people/`.

[Obj11] OBJECT MANAGEMENT GROUP: *Business Process Model and Notation (BPMN)*, 2011. `http://www.omg.org/spec/BPMN/`.

[O'R05] O'REILLY, TIM: *What Is Web 2.0*, 2005. `http://www.oreilly.de/artikel/web20.html`.

[Oul95] OULD, MARTYN A.: *Business Processes: Modelling and Analysis for Re-engineering and Improvement*. Wiley, 1995.

[Pei81] PEIRCE, BENJAMIN: *Linear Associative Algebra*. American Journal of Mathematics, 1881.

[Pel03] PELTZ, CHRIS: *Web Services Orchestration and Choreography*. IEEE Computer Society, Vol. 36, No. 10, 2003.

[Pet62] PETRI, CARL ADAM: *Kommunikation mit Automaten*. Schriften des IMM, 1962.

[PGP00] *Introduction to Cryptography*, 2000. `http://www.pgpi.org/doc/guide/7.0/en/intro/`.

[PL02] PELEGRI-LLOPART, EDUARDO: *JSR 53: JavaTM Servlet 2.3 and JavaServer PagesTM 1.2 Specifications*, 2002. `http://www.jcp.org/en/jsr/detail?id=53`.

[Rad09] RADUL, ALEXEY: *Propagation Networks: A Flexible and Expressive Substrate for Computation*. Doktorarbeit, Massachusetts Institute of Technology, 2009.

[RCF04] ROBINSON, D., K. COAR und THE APACHE SOFTWARE FOUNDATION: *The Common Gateway Interface (CGI) Version 1.1*, 2004. http://www.ietf.org/rfc/rfc3875.txt.

[Res12a] *Restfulie - restful made easy.*, 2012. http://restfulie.caelum.com.br/.

[Res12b] *Restlet - The leading RESTful web framework for Java*, 2012. http://www.restlet.org/.

[Rot08] ROTH, GREGOR: *Server load balancing architectures, Part 1: Transport-level load balancing*, 2008. http://www.javaworld.com/javaworld/jw-10-2008/jw-10-load-balancing-1.html.

[RRY10] RAJAN, KAUSHIK, SRIRAM RAJAMANI und SHASHANK YADUVANSHI: *GUESSTIMATE: A Programming Model for Collaborative Distributed Systems*. PLDI, 2010.

[RSS08] RSS-DEV WORKING GROUP: *RDF Site Summary (RSS) 1.0*, 2008. http://web.resource.org/rss/1.0/spec.

[RW94] RENSINK, AREND und HEIKE WEHRHEIM: *Weak Sequential Composition in Process Algebras*. Concur '94: Concurrency Theory, LNCS 836, Springer-Verlag, 1994.

[Sch99] SCHEER, A.-W.: *ARIS: Business Process Modeling*. Springer-Verlag, 1999.

[Sch02] SCHOLLMEIER, RÜDIGER: *A Definition of Peer-to-Peer Networking for the Classification of Peer-to- Peer Architectures and Applications*. Proceedings of the First International Conference on Peer-to-Peer Computing, 2002.

[Sch09] SCHMIDT, DR. KURT: *Prozess-Optimierung im Output-Management*. Books on Demand GmbH, Norderstedt, 2009.

[Sha09] SHANNON, BILL: *Java EE : XML Schemas for Java EE Deployment Descriptors*, 2009. http://java.sun.com/xml/ns/javaee/.

[SLB09] SHOHAM, YOAV und KEVIN LEYTON-BROWN: *Multiagent Systems: Algorithmic, Game-Theoretic and Logical Foundations*. Cambridge University Press, 2009.

[SMCB96] STEFFEN, BERNHARD, TIZIANA MARGARIA, ANREAS CLASSEN und VOLKER BRAUN: *The METAFrame'95 Environment*. Lecture Notes in Computer Science, Volume 1102, 1996.

[Ste98] STEVENS, W. RICHARD: *UNIX Network Programming, Volume 2: Interprocess Communications (2nd Edition)*. Prentice Hall, 1998.

[Tan07] TANENBAUM, ANDREW S.: *Modern Operating Systems*. Prentice Hall International, 2007.

[vdA00] AALST, W. M. P. VAN DER: *Loosely coupled interorganizational workflows: modeling and analyzing workflows crossing organizational boundaries*. Information and Management 37, 2000.

[vdABV+99] AALST, W.M.P. VAN DER, T. BASTEN, H.M.W. VERBEEK, P.A.C. VERKOULEN und M. VOORHOEVE: *Adaptive Workflow: On the Interplay between Flexibility and Support*. ICEIS, 1999.

[vdAH05] AALST, W. M. P. VAN DER und A. H. M. TER HOFSTEDE: *YAWL: yet another workflow language*. Information Systems, Vol 30, 2005.

[vdAtHKB03] AALST, W.M.P VAN DER, A.H.M. TER HOFSTEDE, B. KIEPUSZEWSKI und A.P. BARROS: *Workflow Patterns*. Distributed and Parallel Databases, 2003.

[Vog08] VOGELS, WERNER: *Eventually Consistent - Revisited*, 2008. `http://www.allthingsdistributed.com/2008/12/eventually_consistent.html`.

[Vog09] VOGELS, WERNER: *Eventually Consistent*. Communications of the ACM, 2009.

[W3C98] *Cool URIs don't change*, 1998. `http://www.w3.org/Provider/Style/URI`.

[W3C05] W3C: *Document Object Model (DOM)*, 2005. `http://www.w3.org/DOM/`.

[W3C08] W3C: *XML Signature Syntax and Processing (Second Edition)*, 2008. `http://www.w3.org/TR/xmldsig-core/`.

[W3C12a] W3C: *HTML*, 2012. `http://www.w3.org/html/`.

[W3C12b] W3C: *XML Essentials*, 2012. `http://www.w3.org/standards/xml/core`.

[Wal07] WALL, LARRY: *Programming is Hard, Let's Go Scripting*, 2007. `http://www.perl.com/pub/2007/12/06/soto-11.html`.

[Wor08] WORKFLOW MANAGEMENT COALITION: *XML Process Definition Language*, 2008. `http://www.wfmc.org/xpdl.html`.

[WSR12] *International Workshop on RESTful Design*, 2012. http://ws-rest.org/.

[WV01] WEIKUM, GERHARD und GOTTFRIED VOSSEN: *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[WWW12] *International World Wide Web Conferences Steering Committee*, 2012. http://www.iw3c2.org/.

[XML02] *Designing Enterprise Applications with the J2EE Platform, Second Edition*, 2002. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/deployment/deployment5.html.

[ZBD11] ZUZAK, IVAN, IVAN BUDISELIC und GORAN DELAC: *Formal Modelling of RESTful Systems Using Finite-State Machines*. ICWE'11 Proceedings of the 11th international conference on Web engineering, 2011.

[zMNS05] MUEHLEN, MICHAEL ZUR, JEFFREY V. NICKERSON und KEITH D. SWENSON: *Developing Web Services Choreography Standards - The Case of REST vs. SOAP*. Decision Support Systems - Special issue: Web services and process management, 2005.

Index