Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. habil. P. Levi
Betreuer:	Dipl.-Inform. R. Lammert
begonnen am:	1. Oktober 2001
beendet am:	31. März 2002
CR-Klassifikation:	D.1.3, I.2.11, I.6.3

Diplomarbeit Nr. 1966

Entwicklung und Implementierung von Multi–Agenten–Szenarien und Simulation des Verhaltens von Agenten beim Zugriff auf gemeinsame Ressourcen

Inna Avroutina

Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Motivation und Zielsetzung

1.1 Motivation

Multi–Agenten–Systeme stellen eine moderne Softwarearchitektur dar, die es ermöglichen soll, komplexe Aufgaben flexibel und robust zu lösen. Multi–Agenten–Systeme werden in verschiedenen Anwendungsbereichen eingesetzt, wie beispielsweise in der Robotik, Fertigung, Verkehrswesen, Telekommunikation, usw.
Für die Entwicklung und Untersuchung von realen Multi–Agenten–Systemen wird oft Simulations–Software eingesetzt. Das liegt zum einen daran, dass man dabei in den ersten Phasen der Entwicklung auf teuere Testläufe verzichten kann. Zum anderen kann man dabei zunächst einige Details außer Acht lassen und auf der konzeptionellen Ebene arbeiten. Daher werden durch die Simulation die Entwicklungskosten reduziert, und nur in der Simulationsphase erfolgreich getestete Agentenarchtekturen müssen in den realen Umgebungen eingesetzt werden. Aus diesen Gründen stellt die Entwicklung von Simulations–Software für Multi–Agenten–Systeme eine wichtige Aufgabe dar.

1.2 Zielsetzung

Im Rahmen der vorliegenden Arbeit wird das Verhalten von Agenten beim Zugriff auf gemeinsame Ressourcen exemplarisch untersucht. Zu diesem Zweck wird ein sich in der Entwicklungsphase befindendes Simulationspaket um einige Komponenten erweitert. Zum einen handelt es sich dabei generell um die Komponenten, die die Kommunikation übers Netz unterstützen.

Zum anderen werden hier einfache Agenten entworfen und implementiert, die im Rahmen spezieller Szenarien koordiniert auf gemeinsame Ressourcen zugreifen. Es werden sowohl zentralistische, als auch verteilte (verhandlungsbasierte) Szenarien untersucht. Die genaue Aufgabenstellung der Arbeit ist in Kapitel 3 beschrieben.

Kapitel 2

Stand der Kenntnisse

In diesem Kapitel sind einige Aspekte aus den Bereichen der Multi–Agenten–Systeme, ihrer Simulation und der Netzwerk–Kommunikation kurz zusammengefasst.

2.1 Multi–Agenten–Systeme

2.1.1 Arbeitsdefinitionen

Agenten und Multi–Agenten–Systeme zählen zu den Begriffen aus dem Bereich der Verteilten Künstlichen Intelligenz (VKI), für die es zur Zeit keine allgemein anerkannten Definitionen existieren.
Unter einem *Agenten* wird in der Regel eine komplexe autonom agierende intelligente Software–Einheit verstanden. Ein *Multi–Agenten–System* (MAS) setzt sich aus mehreren Agenten zusammen, die möglicherweise miteinander kommunizieren, eigene Ziele verfolgen und kooperativ eine globale Aufgabe lösen.
Aus den Definitionen ergeben sich zwei grundsätzliche Fragen:

- wie müssen Agenten aufgebaut sein und
- wie soll ihre Koordination geregelt werden,

damit das Zusammenleben der Agenten möglichst konfliktfrei ist und die gemeinsamen Ziele möglichst effizient erreicht werden.
Diesen beiden Fragen werden die zwei folgenden Abschnitte gewidmet.

2.1.2 Der Autonomiezyklus

Eine mögliche Architektur eines einzelnen Agenten kann durch den Autonomiezyklus ([7]) repräsentiert werden (Abbildung 2.1).

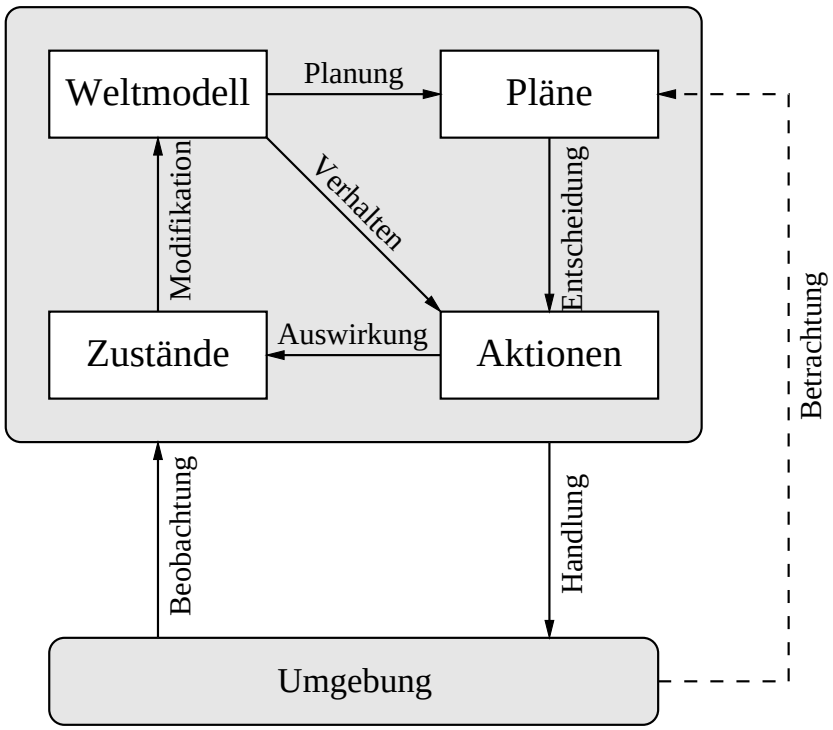


Abbildung 2.1: *Der Autonomiezyklus ([7], [13]).*

Nach dem Autonomiezyklus–Konzept besitzt ein Agent die folgenden vier Komponenten ([17]):

- *Die Weltmodellkomponente* ist die Wissensbasis des Agenten. Hier werden sowohl Informationen über die möglichen Objekte der Umwelt, ihre Zustände und mögliche Aktionen gespeichert als auch über die Ziele des Agenten.
- *Die Plankomponente* enthält alle möglichen Pläne des Agenten.
- *Die Aktionskomponente* enthält alle möglichen Aktionen, die ein Agent ausführen kann.

- *Die Zustandskomponente* enthält alle mögliche externe Zustände der Umwelt.

Diese vier Komponenten sind durch eine Reihe von Funktionen verbunden:

- *Planung:* Der Agent entwirft verschiedene Pläne auf der Grundlage seines Weltmodells und der Zustände der Umwelt.
- *Betrachtung:* Der Agent betrachtet die Umwelt. Dabei kann er evtl. nur einen Teil der externen Zustände wahrnehmen.
- *Entscheidung:* Auf der Grundlage seiner Betrachtung entscheidet sich der Agent für einen Plan und, davon ausgehend, für eine Aktion.
- *Handlung:* Der Agent führt die ausgewählte Aktion aus.
- *Auswirkung:* Der Agent ermittelt die erwartete Veränderung der Umwelt als Auswirkung der Aktion.
- *Beobachtung:* Der Agent überwacht, ob die Aktion erfolgreich war. Es wird die tatsächliche Veränderung der Umwelt beobachtet.
- *Modifikation:* Wenn die Wirkung der Aktion nicht mit der gewünschter übereinstimmt, wird das Weltmodell modifiziert.

2.1.3 Koordination zwischen Agenten

Multi–Agenten–Systeme werden für die Lösung von Aufgaben eingesetzt, die von einem einzelnen Agenten nicht oder nicht effizient genug erledigt werden können. Deswegen stellt die Organisation der Koordination zwischen Agenten einen grundlegenden Aspekt der Konzeption eines MAS dar. Nur dadurch wird es möglich, das System mit den Fähigkeiten auszustatten, die für das Erreichen der gesetzten Ziele notwendig sind.
Man unterscheidet zwischen zwei Arten der Koordination ([16], Abb. 2.2):

- Kooperation: Zusammenarbeit, gemeinsames Verfolgen von Zielen
- Konkurrenz: Verfolgen von unterschiedlichen, sich teilweise widersprechenden Zielen

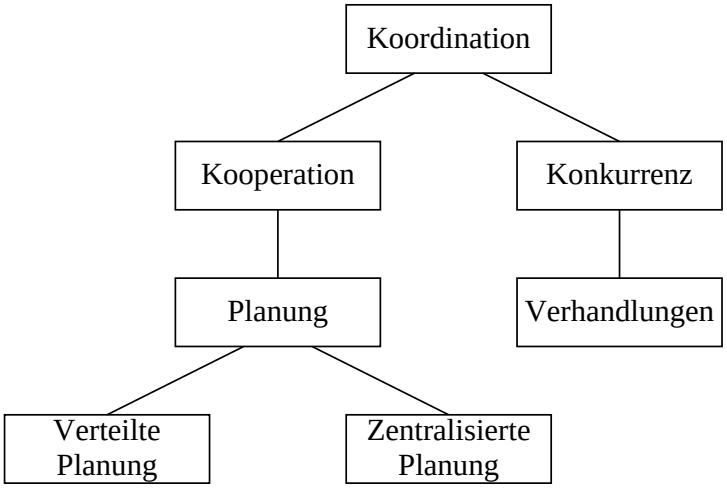


Abbildung 2.2: *Arten der Koordination (nach [16])*

Besonders bei konkurrierenden Agenten müssen Interaktionen koordiniert werden, sofern sie auf die gleichen Ressourcen zugreifen. Das ist wichtig z.B. bei der Zuteilung von Ressourcen, bei der Sequentialisierung des Zugriffs auf Ressourcen, sowie in vielen weiteren Bereichen. Die Koordinationsaufgaben können mit Hilfe unterschiedlicher Ansätze gelöst werden. Insbesondere können sie durch strukturierte, hierarchisch aufgebaute, zentralistische Regelung der Aktivitäten realisiert werden. Eine Alternative dazu stellt die Kommunikationsmöglichkeit dar, bei der die Konversation nicht strukturiert ist, sondern eine flache, gleichberechtigte Verhandlung zwischen verschiedenen Agenten darstellt ([14]).

2.1.4 Verhandlungen

Die Aufgabe, Verhandlungen zwischen Agenten zu realisieren und zu simulieren stellt ein wichtiger Teil der vorliegenden Diplomarbeit dar. Deshalb werden die theoretischen Grundlagen der Verhandlungen in diesem Abschnitt ausführlich beschrieben.

2.1.4.1 Definition und Eigenschaften

Unter einer **Verhandlung** in einem Multi–Agenten–System versteht man den Prozess der Suche von Agenten nach einer gemeinsamen Übereinkunft oder Abmachung ([13]).

Eine Verhandlung kann verschiedene Aspekte beinhalten:

- Reiner Austausch von Informationen
- Neu– bzw. Umverteilung von Aufgaben
- Auflösung von Konfliktsituationen (Ressourcen– bzw. Teilziel–Konflikte) durch:
  - Relaxation von Zielen (Nachgeben)
  - Gegenseitige Zugeständnisse(Konzessionen)
- Täuschungen

2.1.4.2 Das Verhandlungsproblem

Das Verhandlungsproblem kann im allgemeinen in den folgenden drei Gebieten auftreten ([13]):

- *Task Oriented Domain*
- *State Oriented Domain*
- *Worth Oriented Domain*

Bei dem *Task Oriented Domain* sind die Aktivitäten der Agenten durch eine Menge von atomaren Aufgaben (Tasks) beschrieben. Die Ressourcen sind unbeschränkt. Die Aufgaben werden von einem isolierten Agenten ausgeführt, oder sie werden zuerst im gemeinsamen Interesse an mehrere Agenten umverteilt.

Beim *State Oriented Domain* werden die Aktivitäten der Agenten durch Zustandsänderungen beschrieben. Die Ressourcen sind beschränkt. Die Agenten müssen evtl. Konflikte lösen. Die Ziele eines Agenten können durch einen Zustand entweder ganz oder gar nicht erfüllt sein. Für jeden Agenten

existiert eine *boolesche* Zielfunktion.

Beim *Worth Oriented Domain* können die Ziele eines Agenten auch teilweise erfüllt werden. Jedem Zustand des Agenten wird ein reeller Wert zugewiesen. Somit existiert für jeden Agenten eine *reelle* Zielfunktion.

Es gilt im allgemeinen:
Task Oriented Domain $\subset$ *State Oriented Domain* $\subset$ *Worth Oriented Domain*

2.1.4.3 Der Verhandlungsmechanismus

Ein *Verhandlungsmechanismus* ist durch die folgenden vier Bestandteile definiert ([13]):

1. Die *Verhandlungsmenge* definiert alle mögliche Abmachungen, über die verhandelt werden kann.
2. Das *Verhandlungsprotokoll* legt die genauen Regeln für die Verhandlungen fest.
3. Der *Verhandlungsprozeß* legt die genaue Durchführung, d.h. die Reihenfolge und die Häufigkeit der Vorschläge der einzelnen Agenten fest. Kann auch in Punkt 2 mitenthalten sein.
4. Die *Verhandlungsstrategie* legt die regelkonforme individuelle Vorgehensweise der einzelnen Agenten fest.

Bei der Bewertung unterschiedlicher Verhandlungsmechanismen sind u.a. folgende Eigenschaften von Bedeutung:

- Effizienz: keine Ressourcenverschwendung, optimal nach einem gegebenen Optimierungskriterium
- Stabilität: keine Abweichung von einer getroffenen Übereinkunft
- Einfachheit: keinen hohen Rechenaufwand der Agenten bei den Verhandlungen
- Verteilheit: keine zentrale Entscheidungseinheit
- Symmetrie: kein Agent darf von dem Verhandlungsprozess bevorzugt oder benachteiligt werden

Die ausführlichere Beschreibung der Verhandlungen kann [11] entnommen werden.

2.2 Netzwerkkommunikation

Das Thema Netzwerkkommunikation und *CORBA* wurde bereits in mehreren Arbeiten behandelt (siehe z.B. [10], [5]), deshalb wird in diesem Kapitel auf eine detaillierte Beschreibung verzichtet. Innerhalb dieses Kapitels werden die grundlegenden und für diese Diplomarbeit wesentlichen Aspekte basierend auf [10] behandelt.

2.2.1 Die *OMG* und ihre Ziele

Der Entwicklung der Software–Komponenten, die die Netzwerkkommunikation realisieren, liegt im wesentlichen die Arbeit der *Object Management Group* (*OMG*) zugrunde.
Die *OMG* ist ein internationaler, nicht profitorientierter Zusammenschluß von Hardwareherstellern, Softwareentwicklern, Netzwerkbetreibern und kommerziellen Anwendern von Computersystemen. Sie wurde 1989 von elf Unternehmen gegründet und ist zur Zeit das weltweit größte Softwarekonsortium. Im Jahre 2000 bestand die *OMG* aus über 800 Mitglieder ([8]).

Zu den zentralen Aufgaben der *OMG* gehört das Sammeln und Aufarbeiten von technologischem Know–how für die Entwicklung verteilter, aus Objekten (eindeutig identifizierbaren Einheiten) aufgebauter Software [10]. Dieses Wissen stellt die *OMG* ihren Mitgliedern in Form von allgemeinen Richtlinien und industriell verwertbaren Spezifikationen, sowohl von allgemeinen Architekturen als auch von konkreten Komponenten, zur Verfügung.

2.2.2 Die *Object Management Architecture*

Die *Object Management Architecture* (*OMA*) ist die von der *OMG* spezifizierte Softwarearchitektur. Sie ermöglicht die Zusammenarbeit von verschiedenen Anwendungen, unabhängig davon, für welches Betriebssystem bzw. Hardware und mit welcher Programmiersprache sie entwickelt wurden. Wie der Name schon sagt, spielen hier *Objekte* eine wichtige Rolle. Sie sind nämlich die grundlegenden Bausteine für verteilte Anwendungen.

Die für ein Objekt definierten Operationen beschreiben das Verhalten dieses Objekts. Durch sein *Interface* kann man erfahren, welche Operationen das Objekt unterstützt und wie auf das Objekt zugegriffen werden kann. Für die

Beschreibung von Interfaces wird von der *OMG* eine eigene Sprache *Interface Description Language (IDL)* zur Verfügung gestellt. In dieser Sprache werden die von außen sichtbaren Eigenschaften von Objekten (Attribute, Operationen, Exceptions, Datentypen, die für die Kommunikation mit einem Objekt benötigt werden) spezifiziert.

Ein Objekt kann in einer der bekannten Programmiersprachen, wie z.B. *C*, *C++*, *Smalltalk*, *Ada*, *Java*, usw., implementiert werden. Für jede dieser Sprachen ist von der *OMG* ein Zusatzdokument, das *Language–Mapping*, definiert. In dem steht, wie die in IDL beschriebenen Interfaces und Datentypen in Konstrukte der jeweiligen Programmiersprache umgesetzt sind und wie in dieser Programmiersprache der Aufruf einer Operation aussieht.

Die OMA unterteilt die Bestandteile einer verteilten Anwendung in Komponenten. Eine Komponente besteht aus einer Menge von Objekten, die gemeinsam eine bestimmte Aufgabe (Dienst bzw. Service genannt) erfüllen.

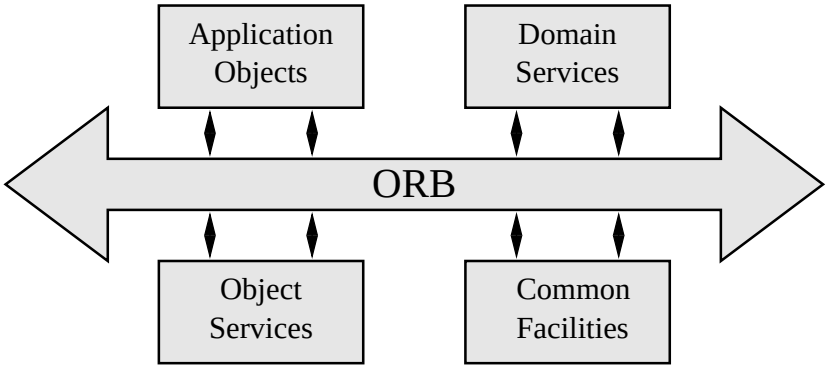


Abbildung 2.3: *Object Management Architecture*

Jede Komponente wird von der OMA einer der fünf Gruppen zugeordnet (Abb. 2.3). Im folgenden werden diese Komponenten kurz beschrieben.

- *Object Request Broker (ORB)* ist eine Komponente, die von Objekten zur Kommunikation mit anderen Objekten sowohl innerhalb desselben Programms als auch mit Objekten in einem anderen Programm benutzt werden kann. Der Aufruf einer Operation geschieht für den

Aufrufer transparent, d.h. weder der Aufenthaltsort des Objektes noch das Betriebsystem, das verwendet wird, noch die Programmiersprache, in der das Objekt geschrieben ist, müssen dem Aufrufer bekannt sein.

- *Object Services* sind die aufgabenbereich–unabhängige standardisierte Schnittstellen, die von mehreren verteilten Anwendungen benutzt werden können. Sie gelten als Teil der verteilten Infrastruktur, die von allen Objekten genutzt werden kann.

  Von der OMG werden Object Services für z.B. folgende Aufgaben definiert:

  - *Naming Service*: das Auffinden von Objekten im Netz
  - *Event Service*: die Behandlung asynchroner Ereignismeldungen
  - *Persistent Object Service*: die langfristige Speicherung von Objektzuständen
  - *Security Service*: der Schutz des Systems vor unerlaubten Benutzung
  - *Time Service*: die Synchronisation von Uhren

- *Application Objects* werden für verschiedene konkrete Anwendungen entwickelt. Sie sind von der OMG nicht standardisiert.

- *Domain Services*, die auch *Vertical Market Facilities* genannt werden, sind standardisierte Lösungen für spezielle Anwendungsgebiete wie z.B. Finanzwesen, Fertigung, Gesundheitswesen und Telekommunikation.

- *Common Facilities*, die auch *Horizontal Common Facilities* bezeichnet werden, sind komplexe funktionale Einheiten, die durch Konfigurierungsmechanismen an verschiedene Aufgabenstellungen angepasst und somit in mehreren Anwendungen eingesetzt werden. Die Common Facilities bauen auf den Object Services auf und sind stärker spezialisiert als diese.

  Als Beispiele von Common Facilities können Dienste zur Druckersteuerung, zur Dokumentenverwaltung, Datenbanken, E–Mail, sowie grafische Benutzeroberflächen genannt werden.

2.2.3 *CORBA*

Die *Common Object Request Broker Architecture* (*CORBA*) ist der OMG–Standard, der den Aufbau des ORB, seine Bestandteile sowie deren Verhalten und Interfaces beschreibt. Zu den Vorteilen dieses Standardes zählen unter anderem seine Offenheit und Portabilität. Es existieren Implementierungen von CORBA für die meisten gängigen Hardwareplattformen, Betriebssysteme und Programmiersprachen.

2.2.4 Namensdienst

CORBA–Objekte können prinzipiell über das gesamte Netzwerk verteilt werden. Eine der zentralen Fragestellungen bei der Entwicklung von verteilten Anwendungen ist die Frage, wie die verteilten Komponenten einander finden. Zur Lösung dieser Frage wird in CORBA der *Namensdienst (Naming Service)* spezifiziert.
In den folgenden Abschnitten wird beschrieben, wie von einem Klienten ein bestimmtes Objekt aufgefunden werden kann.

2.2.4.1 Grundidee

CORBA–Objekte werden durch Objekt–Referenzen weltweit eindeutig identifiziert. Objekt–Referenzen werden von dem ORB mit Hilfe eines *Objekt Adapters* vergeben. Bei vielen CORBA–Plattformen geschieht dies automatisch bei der Erzeugung einer Instanz der Objekt–Implementation.
Die Vermittlung von Objekt–Referenzen an Klienten erfolgt nach dem CORBA–Design nicht direkt durch den ORB, sondern sie wird einem separaten Dienst, dem **Namensdienst** überlassen. Bevor der Klient diesen Vermittlungsdienst benutzen kann, benötigt er eine Objekt–Referenz für das entsprechende CORBA–Objekt, um den Dienst zugänglich zu machen. Mit der Funktion

```
resolve_initial_references(String id);
```

die vom ORB bereitgestellt wird, kann ein Klient für eine sehr geringe Anzahl grundlegender CORBA–Objekte die zugehörigen Objekt–Referenzen ermitteln. Eins davon ist der *Namensdienst*. Als Argument ist in diesem Fall die vordefinierte Konstante `NameService` anzugeben.

Server können bei einem Namensdienst die von ihnen bereitgestellten Objekte unter einem Namen registrieren lassen. Klienten können anschließend unter Angabe eines Namens die zugehörige Objekt–Referenz erfragen, ohne wissen zu müssen, wo sich das zugehörige Objekt befindet. Die Organisation der Namensvergabe erfolgt durch den Programmierer.
In der Abbildung 2.4 ist die Vorgehensweise bei der Vermittlung eines Objektes dargestellt.

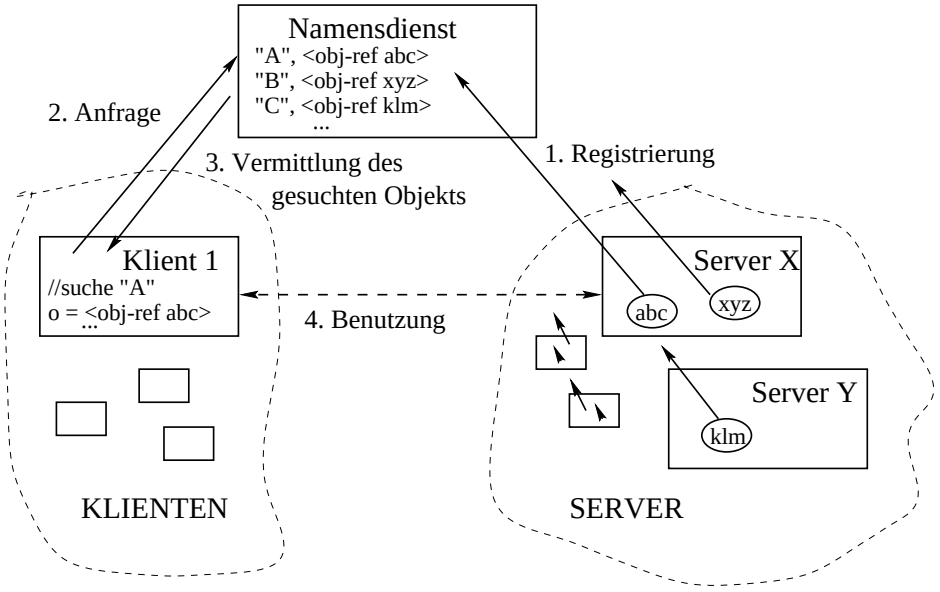


Abbildung 2.4: *Namensdienst (nach [10])*

Bei der Installierung der CORBA–Plattform wird von der Systemadministration festgelegt, ob jeder Anwendung ein eigenes Exemplar des Namensdienstes bereitgestellt wird, oder eine Reihe von Anwendungen auf einem Rechner oder innerhalb eines Netzwerkes einen gemeinsamen Namensdienst–Server benutzen.

2.2.4.2 *COSS–Naming*

COSS–Naming ist die Spezifikation des Dienstes für die Vermittlung von Objekt–Referenzen an Klienten, die von der OMG vorgesehen ist. Die zen-

tralen Begriffe für *COSS–Naming* sind *Namensbindung* und *Namenskontext*.

Eine *Namensbindung* ist eine Zuordnung von einem Namen zu einem CORBA–Objekt, also ein Paar {*Name*, *Objekt*}. Mehrere Namensbindungen können zu einem *Namenskontext* zusammengefasst werden. In einem Namenskontext sind alle Namen eindeutig, d.h. in ihm gibt es keine zwei Namensbindungen mit identischen Namen. Es ist jedoch zulässig, dass in einem Namenskontext verschiedenen Namen dasselbe Objekt zugeordnet wird (*Aliasing*).

Namenskontexte bieten Operationen zum Einfügen neuer Namensbindungen (*Binden des Objekts an einen Namen*) und zum Entfernen von Namensbindungen an. Unter dem *Auflösen eines Namens* versteht man das Ermitteln desjenigen Objekts innerhalb eines Namenskontexts, dem mittels einer Namensbindung der jeweilige Name zugeordnet wurde. Da demselben Namen in einem anderen Namenskontext ein anderes oder gar kein Objekt zugeordnet sein kann, ist es sinnvoll, den Namenskontext anzugeben, wenn es ums Binden oder Auflösen eines Namens geht.

2.3 Softwarepaket zur Simulation von Agentengruppen

Das der vorliegenden Diplomarbeit zugrundeliegende Softwarepaket zur Simulation von Agentengruppen wurde in der Abteilung Bildverstehen der Universität Stuttgart im Zusammenhang mit dem Projekt „Soziale Strukturen in Gruppen intelligenter, multimotivierter, emotionaler Agenten und deren Entstehung“ entwickelt ([1]).

2.3.1 Ereignis–Konzept und Kommunikationskanäle

Die entwickelte Simulationsumgebung wurde als ereignisbasierte Anwendung konzipiert. Es ist gelungen, dieses klassische Konzept so abstrakt zu formulieren, dass nicht nur der bloße Nachrichtenaustausch, sondern auch andere Interaktionen zwischen Agenten bzw. zwischen Agenten und ihrer Umgebung, wie z.B die taktile Wahrnehmung, die Bewegung der dynamischen Objekte im Raum (sowohl der Agenten als auch die der anderen beweglichen Einheiten), ereignisorientiert modelliert werden können.

Die Ereignisse sind in diesem Kontext durch folgende Merkmale charakterisiert:

- Es sind diskrete, in sich abgeschlossene Einheiten.
- Es handelt sich dabei um abstrakte Objekte, die bestimmte Informationen beinhalten.
- Es sind Objekte, die einen kontrollierten Zugriff auf die Daten unterstützen (dies ist insofern wichtig, weil die Wartbarkeit des Systems dadurch erhöht wird).

Durch die Verwendung des Ereignis–Konzeptes können Erweiterbarkeit, Skalierbarkeit und Überwachbarkeit des Systems verbessert werden.

Das Ereigniskonzept stellt eine Abbildung des bewährten und praktisch einsetzbaren Agentenmodells in einer Simulationsumgebung dar. Von den fünf wesentlichen Komponenten eines Agenten (Sensorik, Weltmodell, Entscheidungseinheit, Planungseinheit und Aktorik) sind drei (Weltmodell, Entscheidungseinheit, Planungseinheit) reine Software–Komponenten

und zwei (Sensorik und Aktorik) mehr an die Hardware gebunden. Das Ereigniskonzept repräsentiert in dem Simulationsmodell gerade diese zwei Komponenten, wobei das Senden eines Ereignisses der Aktorik eines autonomen Agenten und das Empfangen eines Ereignisses der Sensorik entspricht.

Das Konzept der Kommunikationskanäle wird sehr breit gefaßt. Alle Wahrnehmungssinne eines Agenten können als Empfänger an jeweils einem Kommunikationskanal betrachtet werden.

2.3.2 Ereignisbasierte Bewegung von Objekten

Die Verwendung des Ereignis–Konzeptes für die Simulation der Bewegung von Objekten in einer dynamischen Umgebung erlaubt eine einheitliche und für einen äußeren Beobachter übersichtliche Darstellung aller Phänomene, die bei der Bewegung eine Rolle spielen. Demnach wird dem Raum, in dem sich die dynamischen Objekte bewegen sollen, ein Kommunikationskanal zugeordnet, an dem alle mobilen Objekte angedockt sind. Zu beachten ist, dass einem Objekt seine eigenen Koordinaten nicht unbedingt bekannt sein müssen um sich bewegen zu können. Um sich in Bewegung zu setzen, muss ein Agent die Bewegungsrichtung angeben und eine gewisse Energie für die Ausführung der Aktion bereitstellen. Ein derartig formulierter Bewegungswunsch wird in Form eines Bewegungsereignisses (Move–Event) umgesetzt, das über den Bewegungskanal an alle im Raum existierenden – d.h. an dem Kanal angemeldeten – Objekte weitergeleitet wird, die dem sich bewegenden Objekt unterwegs begegnen (Kollision). Diese ziehen dann den Teil der bereitgestellten Energie ab, der erforderlich ist, um sie zu passieren (Falls die Energie nicht ausreicht, wird die gesamte bereitgestellte Energie abgezogen). Dieses Verhalten der Objekte hat unterschiedliche Interpretationen. An der Oberfläche wird z.B. immer Energie abgezogen, was einer einfachen Reibung entspricht. Es existieren aber auch feste Gegenstände, durch die keine Bewegung möglich ist. Diese sind dadurch charakterisiert, dass sie die gesamte Bewegungsenergie absorbieren, ohne den Agenten weiter in die gewünschte Richtung durchzulassen. Somit endet die Bewegung eines Agenten entweder neben einem solchen Objekt (die Agenten sollen im Laufe der Zeit lernen, dies zu vermeiden) oder an einer neuer Position. Anzumerken ist, dass die Energie im System sich dissipativ verhalten muss. Zudem muss die simulierte Welt über eine wohldefinierte Oberfläche verfügen.

Kapitel 3

Aufgabenstellung

Gegenstand dieser Arbeit ist die Erweiterung eines bereits bestehenden Softwarepaketes zur Simulation von Agentengruppen (Kapitel 2.3). In einer ersten Entwicklungsphase soll eine geeignete Implementierung einer Kommunikationsschicht auf Basis der CORBA–Architektur entstehen (Kapitel 2.2), um die Möglichkeit zur Interprozesskommunikation zu gewährleisten. Dabei soll die bereits existierende Architektur um weitere Objekte erweitert werden, so dass bestehende Schnittstellen nicht geändert werden müssen. Das Ziel ist also eine möglichst effektive und transparente Integration der zu entwickelnden Komponenten in das existierende Konzept.

Im zweiten Teil der Arbeit soll das Kommunikations– und Problemlöseverhalten einer Gruppe von Agenten am Beispiel eines Multi–Agenten–Szenarios geprüft werden. In diesem Szenario befinden sich die einzelnen Agenten in einem Konkurrenzkampf um gemeinsame Ressourcen. Zum Zwecke der Konfliktlösung sollen verschiedene Zugriffsstrategien entwickelt, implementiert und untersucht werden. Dabei sollen sowohl zentralisierte als auch verteilte Algorithmen eingesetzt werden, wobei im verteilten Fall die Autonomie der Agenten erhalten werden soll. Beim dezentralisierten Ansatz sollen die Agenten also gemeinsam eine Lösung erarbeiten, wobei sie zu diesem Zwecke Verhandlungen führen müssen.

Ein wichtiger Aspekt dieser Arbeit betrifft die Implementierung der Agenten als eigenständige Threads, die miteinander und mit ihrer Umgebung interagieren müssen. Dabei können Synchronisationsprobleme entstehen, die dann gelöst werden müssen. Die Bewegungen (Aktionen) der einzelnen Agenten

sollen simuliert und visualisiert werden, wodurch sich bestimmte Effizienzanforderungen an die zu entwickelnden Komponenten ergeben.

Kapitel 4

Kommunikation der Agenten über das Netz

Die ursprüngliche Implementierung von Multi–Agenten–Systemen im Rahmen des Softwarepaketes zur Simulation von Agentengruppen fand auf einem Rechner statt. Da aber mit der wachsenden Anzahl der Agenten und steigernder Komplexität der implementierten Algorithmen der für die Simulation benötigte Rechenaufwand steigt, wird als eine Lösung für die Entlastung des Systems die Verteilung der Simulation auf mehreren Rechner vorgeschlagen. Einem oder mehreren Agenten wird dann ein Rechner zur Verfügung gestellt, und die Kommunikation zwischen Agenten erfolgt über das Netz.

4.1 Lösungsansatz

Laut der Aufgabenstellung (Kapitel 3), ist bei der Implementierung der Kommunikationsschicht besonders darauf zu achten, dass neu zu entwickelnde Komponenten keine Änderung der bereits bestehenden Architektur verursachen, so dass die bestehenden Schnittstellen nicht geändert werden müssen. Die Architektur des Softwarepaketes zur Simulation von Agentengruppen bietet die Möglichkeit, dieser Anforderung gerecht zu werden, da sie weitgehend auf dem Konzept der Kommunikationskanäle basierend aufgebaut ist. Alle Objekte (sowohl Agenten als auch weitere Gegenstände der Simulation) kommunizieren miteinander über Kommunikationskanäle, wobei die Einzelheiten ihrer Implementierung nicht von Bedeutung sind. Nach dem in Abschnitt 2.3 beschriebenen Ereignis–Konzept erfolgt das Empfangen und das

Bearbeiten eines Ereignisses bei einem Receiver (Empfänger) durch den Aufruf der Routine `void handleIncomingEvent (Event e, Connection cn)`. Damit die Kommunikation übers Netz erfolgen kann, ist nun lediglich eine Implementierung der neuen Komponenten auf der Sender– und Receiverseite notwendig, die den Aufruf dieser Routine übers Netz unterstützen.
Zu diesem Zweck ist ein CORBA–Interface zu definieren und zu implementieren, das den Aufruf dieser Routine auf der Reseiver–Seite ermöglicht. Für das Auffinden von Agenten (Receiver) im Netz kann der im Abschnitt 2.2.4 beschriebene Namensdienst benutzt werden.

4.2 Auswahl der CORBA–Entwicklungsumgebung

Das Paket für die Simulation der Agenten ist in der Programmiersprache *Java* implementiert, die Kommunikation zwischen Agenten erfolgt auf Basis der CORBA–Architektur. Als CORBA–Entwicklungsumgebung wurde *ORBacus* (entwickelt von der Firma *IONA Technologies*) benutzt. Zum einen wurde die Auswahl auf Grund der freien Verfügbarkeit dieser CORBA–Entwicklungsumgebung getroffen. Zum anderen wurde ORBacus gewählt, weil diese Entwicklungsumgebung den *java development toolkit* JDK1.2, mit dem das Softwarepaket für die Simulation der Agenten implementiert ist, unterstützt. Bei der anderen freien CORBA–Entwicklungsumgebung, *OrbixWeb*, die für die vorliegende Arbeit prinzipiell auch in Frage käme, ist das nicht der Fall.
Ohne auf die Details einzugehen, können die wichtigste Eigenschaften der ORBacus –Entwicklungsumgebung folgendermassen zusammengefasst werden ([3]):

- vollständige CORBA IDL Unterstützung
- C++ und Java Language Mappings
- einfache Konfiguration und Bootstrapping
- Portable Objekt Adapter
- Werte–Semantik bei Objekten (Objects by Value)
- Portable Interzeptoren

- Single– und Multi–Threading
- Active Connection Management
- Fault Tolerance Extensions
- Dynamic Invocation und Dynamic Skeleton Interface
- Any–Objekte (Dynamic Any)
- Interface und Implementation Repository
- dynamisch einsetzbare Protokolle (Pluggable Protocols)
- IDL zu HTML und IDL zu RTF Dokumentationswerkzeuge
- Naming, Event, Time und Property Services

4.3 Entwurf und Implementierung

Bei der Implementierung dieses Teils der Diplomarbeit wurde innerhalb des Paketes *MAS* zur Simulation von Multi–Agenten–Systemen ein neues Paket *NETCOM* entwickelt, das die verteilte Kommunikation basierend auf dem CORBA–Standard ermöglicht.
Nach dem in Abschnitt 4.1 beschriebenen Lösungsansatz wird der für die Netz–Kommunikation notwendige CORBA–Interface `rcv.idl` folgendermaßen definiert:

```
typedef sequence<octet> EventSeq;

interface rcv
{
  void handleEvent (in EventSeq evSeq);
};
```

Die Implementierung dieses Interfaces `rcvimpl.java` ermöglicht es, die Routine `void handleIncomingEvent (Event e, Connection cn)` des Receivers transparent für den Sender aufzurufen.
Die implementierte CORBA–Kommunikationschnittstelle wird innerhalb der im Rahmen dieser Arbeit realisierten Klassen

```
public class NetSender

public class NetReceiver
```

verwendet.
Bei der verteilten Simulation werden Objekte der Klassen `Sender` und `Receiver` durch Objekte der Klassen `NetSender` und `NetReceiver` erweitert. Die neue Architektur der Kommunikationsschicht ist in der Abbildung 4.1 dargestellt. Dabei ist zu beachten, dass ein Receiver sowohl ein Agent als auch ein Kanal sein kann.

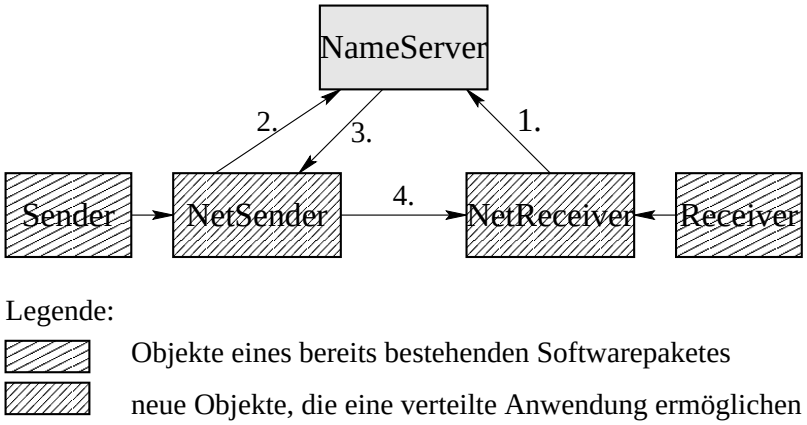


Abbildung 4.1: *Sender–Receiver–Protokol*

Erläuterung:

1. NetReceiver registriert den Receiver beim NameServer. Der NameServer bildet zu dem Namen des Receivers eine Objekt–Referenz.

2. Der NetSender erfragt die Objekt–Referenz zu dem Namen des Receivers, dem der Sender eine Nachricht schicken will.

3. Die Objekt–Referenz wird ermittelt.

4. Eine Nachricht wird an den Receiver verschickt.

Um Chaos im Namensraum zu vermeiden, wird für die Anwendung ein Kontext *BV.DA* angelegt. Weitere Kontexte bzw. Objekte sollen unter

diesem Kontext angelegt werden.

Ein Objekt der Klasse `NetReceiver` hat die folgende Funktionalität:

- initialisiert ORB,
- erzeugt ein Receiver–Objekt der Klasse rcvimpl und
- registriert dieses Objekt im Namensraum (Kontext) *root.BV.DA*

Bei dem Registrieren versucht der NetReceiver, den Kontext *BV.DA* relativ zum Root–Kontext zu ermitteln. Falls dieser Kontext noch nicht existiert (das ist der Fall, wenn der NetReceiver zum ersten Mal gestartet wird) oder falls er unvollständig ist, werden die Kontexte *root.BV* und/oder *root.BV.DA* angelegt. Das Receiver–Objekt wird dann im Kontext *root.BV.DA* registriert.

Ein Objekt der Klasse `NetSender` hat die folgende Funktionalität:

- initialisiert ORB,
- ermittelt benötigte NetReceiver–Referenz,
- verschickt eine Nachricht an den NetReceiver.

Um den Nameserver zu starten, soll im Verzeichnis `/MAS` das folgende Kommando angegeben werden:

```
java com.ooc.CosNaming.Server -i >& NETCOM/filename
```

Die Adresse des Nameservers wird damit in einer Datei abgelegt, die von NetSender und NetReceiver später eingelesen wird.

Zusammenfassend läßt sich festhalten, dass im Rahmen dieses Teils der Arbeit die Möglichkeit der Anwendung der CORBA–Architektur für die verteilte Agenten–Simulation in dem vorhandenen Softwarepaket zur Simulation von Agentengruppen getestet wurde. Die Netz–Kommunikationskomponenten konnten entwickelt, implementiert und in das vorhandene Softwarepaket integriert werden, ohne eine Änderung der bereits bestehenden Architektur zu verursachen.

Kapitel 5

Kampf um gemeinsame Ressourcen: Entwurf

In diesem Kapitel werden das Simulationsszenario, die Auswahlmöglichkeiten einer Energiequelle, verschiedene Algorithmen für den Zugriff auf diese Quelle und entsprechende Arten von Agentenverhandlungen beschrieben.

5.1 Szenario

5.1.1 Simulationsumgebung

Im Rahmen des untersuchten Szenarios wird eine Umgebung betrachtet, in der sich mehrere Agenten bewegen. Die Aufgaben, welche die Agenten in der Umgebung erledigen, ihre Ziele, etc., werden in der vorliegenden Arbeit nicht betrachtet. Es wird lediglich angenommen, dass Agenten sowohl für diese Aufgaben als auch allgemein für die Bewegung Energie benötigen. Innerhalb der Umgebung befinden sich eine bzw. mehrere Energiequellen, die von allen Agenten zur Auffrischung von Energievorräten benutzt werden können.
Die Größe der Umgebung, die Anzahl der Energiequellen und die Lage der Energiequellen in der Umgebung kann beliebig sein, wird jedoch innerhalb eines Simulationslaufs als konstant angenommen.

5.1.2 Wissen eines Agenten

Damit ein Agent Entscheidungen über seine Aktivitäten treffen kann, muss er über eine bestimmte Wissensbasis verfügen. Im Rahmen der vorliegenden

Diplomarbeit wird angenommen, dass jedem Agenten das folgende Wissen zur Verfügung steht:

- aktuelle Koordinaten des Agenten
- aktueller Zustand des Agenten (insbesondere sein aktueller Energievorrat)
- Lage aller Energiequellen
- aktueller Zustand der Energiequelle, bei der der Agent tanken will

Dem Agenten sind weder die Koordinaten anderer Agenten, noch ihre Zustände bekannt.

5.1.3 Ablauf

Am Anfang der Simulation sind Agenten in der Umgebung verteilt. Jeder Agent besitzt einen anfänglichen Energievorrat. Im Laufe der Simulation bewegen sich die Agenten in der Umgebung und verlieren dabei Energie. Wenn bei einem Agenten ein vordefiniertes Minimum des Energievorrates erreicht ist, muss der Agent zur Energiequelle gehen, um seinen Energievorrat aufzufrischen. Zu jedem Zeitpunkt darf jeweils nur ein Agent Energie an der Quelle tanken. Erst wenn er fertig ist, kann ein anderer Agent die Zugriffsrechte auf die Quelle erhalten. Die Agenten verlieren Energie auch beim Warten.

5.1.4 Aktionen eines Agenten

Entsprechend dem in Abschnitt 2.1.2 beschriebenen Autonomiezyklus eines Agenten, besitzt der Agent eine Aktionskomponente, die alle möglichen Aktionen dieses Agenten beschreibt. Unabhängig davon, welche Variante der Koordination beim Zugriff auf eine Energiequelle benutzt wird, kann ein Agent eine der sechs grundlegenden Aktionen ausführen:

1. Einfache Bewegung
2. Bewegung zur Energiequelle
3. Umgehen eines Hindernisses

4. Warten
5. Energie tanken
6. Verlassen der Quellenumgebung
7. Sterben

Im Weiteren werden diese Aktionen ausführlicher beschrieben.

Einfache Bewegung:
Ein Agent bewegt sich in der Simulationsumgebung. Es wird dabei angenommen, dass der Agent während dieser Phase bestimmte Aufgaben erledigt, die im Rahmen der vorliegenden Arbeit nicht modelliert werden. Die Richtung seiner Bewegung hat aus diesem Grund keine Bedeutung für die vorliegende Arbeit und wird deshalb zufällig gewählt. Wenn der Agent mit einem anderen Agenten zusammenstößt bzw. gegen ein Hindernis (eine Energiequelle oder eine Wand) stößt, wird die Richtung neu berechnet.
Bei jedem Schritt verliert der Agent einen Anteil des Energievorrates. Bei einem Zusammenstoß geht noch zusätzlich Energie verloren.

Bewegung zur Energiequelle:
Während der obengenannten einfachen Bewegung überprüft jeder Agent, ob er noch über genug Energie für die weitere Bewegung verfügt. Wird der Energievorrat kleiner als ein vorgegebenes Minimum, bewegt sich der Agent in Richtung der Energiequelle, um seinen Energievorrat aufzufrischen.
Wenn mehrere Energiequellen vorhanden sind, muss der Agent eine Quelle auswählen. Einige Möglichkeiten der Auswahl einer Energiequelle sind in Abschnitt 5.2 ausführlich beschrieben. In der jetzigen Realisierung wird vom Agenten die Entfernung zu jeder Energiequelle berechnet. Danach geht der Agent zu der Quelle, die am geringsten entfernt ist.

Umgehen eines Hindernisses:
Wenn dem Agenten, der die Aktion *Bewegung zur Energiequelle*, *Tanken* oder *Verlassen der Quellenumgebung* ausführt, ein anderer Agent (oder ein Hindernis) im Wege steht, muss ein Hindernisvermeidungsalgorithmus eingeschaltet werden. Ein einfacher, jedoch effizienter Ansatz dafür besteht darin, dass der Agent das Hindernis umgeht, indem er einige Schritte tangential zu seiner bisherigen Bewegungsrichtung geht und danach das

frühere Ziel wieder verfolgt (die Richtung wird dabei neu berechnet).

Warten:
Wenn der Agent die Umgebung der Energiequelle erreicht hat, muss er evtl. warten. Die konkrete Realisierung dieser Aktion hängt stark von dem benutzten Algorithmus des Zugriffes auf gemeinsame Ressourcen ab. Deshalb kann sie nicht generell implementiert werden und wird später für jeden Algorithmus ausführlich beschrieben.

Verlassen der Quellenumgebung:
Wenn der Agent getankt hat, muss er die Umgebung der Energiequelle verlassen. Dabei wird er aus der Warteliste entfernt und kann sich weiter frei bewegen.

Sterben:
Wenn ein Agent seine gesamte Energie verbraucht hat, „stirbt“ er. Falls der Agent in der Warteliste eingetragen ist, wird sein Eintrag gelöscht. Ab dem Zeitpunkt ist der gestorbene Agent für die anderen Agenten nicht mehr sichtbar, d.h. es gibt mit ihm keine Zusammenstöße mehr.

Ausser den hier beschriebenen grundlegenden Aktionen eines Agenten beim Zugriff auf gemeinsame Ressourcen können noch algorithmenspezifische Aktionen dazukommen, abhängig davon, welcher Algorithmus für die Agentenkoordination angewendet wird. Diese Aktionen werden zusammen mit den entsprechenden Algorithmen ausführlich beschrieben.

5.2 Auswahl einer Energiequelle

Für den Fall, wenn in der Simulationsumgebung mehrere Energiequellen vorhanden sind, muss sich ein Agent, der tanken will, für eine der Quellen entscheiden. Dabei kann er folgende **Entscheidungskriterien** berücksichtigen:

- Entfernung zu den einzelnen Energiequellen
- Anzahl der Einträge in den jeweiligen Wartelisten
- Beide der obengenannten Kriterien gleichzeitig

Bei dem letzten Kriterium ist es sehr wichtig, das richtige Verhältnis zwischen den beiden Parametern zu finden. Ein Agent soll überlegen, ob sich der weite Weg zu der Energiequelle mit der kleineren Warteliste, bei dem er mehr Energie verliert, lohnt. Möglicherweise ist es vernünftiger, den kürzeren Weg zu wählen und danach länger zu warten. Solche Entscheidungen können von einem Agenten anhand des Wissens über die Größe der Energieanteile, die während des Bewegens und des Wartens verloren gehen, gemacht werden.

Es existiert aber auch das Risiko, dass zu dem Zeitpunkt, als der Agent die entfernte Energiequelle mit der kleineren Warteliste erreicht hat, bei der Quelle sich schon weitere Agenten angemeldet haben, d.h. die Warteliste ist größer geworden. Das Risiko kann jedoch nicht vermieden werden, sofern dem Agenten die Koordinaten und Zustände anderer Agenten, oder i.a. ihre Pläne, nicht bekannt sind.

Es besteht auch die Möglichkeit, eine Energiequelle einmal fest für alle weiteren Tankvorgänge auszuwählen. Dabei entscheidet sich ein Agent anhand eines der obengenannten Kriterien für eine Energiequelle und benutzt sie jedesmal zum Tanken. Eine Alternative wäre, sich während der Bewegung zur Quelle bzw. des Wartens, über die aktuelle Situation zu erkundigen, um sich evtl. für eine andere Energiequelle zu entscheiden.

Eine Erweiterung des beschriebenen Szenarios kann beispielsweise darin bestehen, dass die Koordinaten der Energiequellen den Agenten nicht von Anfang an bekannt sind, sondern erst im Laufe der Simulation ermittelt werden sollen. Ein Agent kann dann die Lage einer Quelle, sobald er sie gefunden hat, in sein Weltmodell übernehmen. Daraufhin kann er über diese Information frei verfügen, d.h. entweder nur für sich selbst behalten oder sie auch den anderen Agenten mitteilen.

5.3 Zentralisierter Algorithmus

Bei dem zentralisierten Algorithmus werden die Zugriffe auf eine Energiequelle von einem Wächter kontrolliert. Wenn ein Agent an der Quelle Energie tanken will, muss er sich beim Wächter anmelden. Wenn die Quelle frei ist, wird dem Agenten die Erlaubnis zum Tanken erteilt. Wenn die Quelle bereits von einem anderen Agenten besetzt ist, wird der Agent in die Warteliste

eingetragen, die vom Wächter verwaltet wird.
Wenn ein Agent die Quelle verlässt, meldet er sich beim Wächter ab. Der Wächter kann dann dem ersten in der Warteliste eingetragenen Agenten die Erlaubnis zum Energietanken erteilen. Davor wird der Eintrag für diesen Agenten aus der Warteliste entfernt (Abb. 5.1).

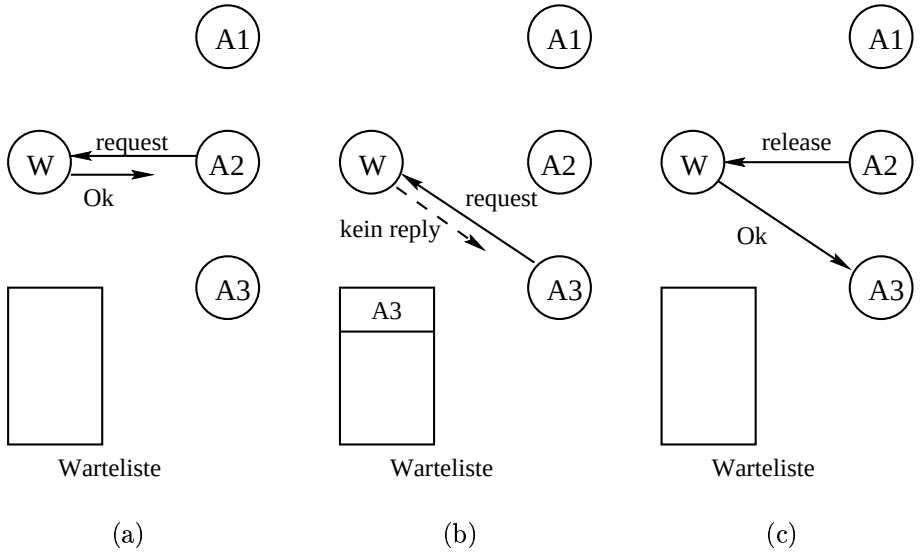


Abbildung 5.1: *Wächter–Agenten–Protokoll*

Erläuterung:

a) Der Agent A2 bittet den Wächter (W) um Erlaubnis, Energie an der Energiequelle zu tanken. Die Erlaubnis wird erteilt.

b) Der Agent A3 bittet auch um Erlaubnis, Energie zu tanken. Der Wächter antwortet nicht (Agent A3 wird blockiert) und trägt den Agenten A3 in die Warteliste ein.

c) Wenn der Agent A2 die Quelle verlässt, informiert er den Wächter, der dann dem Agenten A3 die Erlaubnis erteilt.

Die Wächter–Aufgaben können z.B. von der Energiequelle bzw. von einem Agenten (im Weiteren Agenten–Koordinator genannt) übernommen werden. Durch den Aufbau der Warteliste entscheidet der Wächter über die Rei-

henfolge des Zugriffes auf die Energiequelle durch die bei ihm angemeldeten Agenten. Dabei können verschiedene Strategien des Aufbaus und der Verwaltung der Warteschlange benutzt werden. Sie werden im nächsten Abschnitt beschrieben.

5.3.1 Möglichkeiten des Aufbaus der Warteschlange

Es existieren mehrere Möglichkeiten der Aufbau einer Warteschlange beim zentralisierten Algorithmus für den Zugriff auf gemeinsame Ressourcen.

1. **Einfache Warteschlange (FIFO):**

   Die Agenten werden von dem Wächter der Reihe nach in die Warteliste eingetragen. Der Agent, der als erster kommt, wird als erster mit Energie versorgt.

   Die Warteliste soll in diesem Fall nur Agentennamen als Einträge enthalten.

2. **Von der Restenergie der Agenten abhängige Warteschlange:**

   Die Reihenfolge der Einträge in der Warteliste ist von dem restlichen Energievorrat der Agenten abhängig. Die mit wenig Energie zu der Quelle gekommenen Agenten werden in die oberen Positionen der Warteliste plaziert und schneller als die anderen mit Energie versorgt.

   Für jeden Agenten wird in der Warteliste ein Eintrag, der aus dem Agentennamen und dem restlichen Energievorrat besteht, gespeichert.

3. **Durch mehrere Kriterien bestimmte Warteschlange**

   Der Wächter kann die Warteliste nach mehreren Kriterien aufbauen. Solche Kriterien können z.B. folgende sein (hierarchisch absteigend):

   - Agentenpriorität
   - Restenergievorrat eines Agenten
   - Zeitpunkt der Anmeldung beim Wächter

Die beschriebenen Strategien des Aufbaus von Warteschlangen kommen in verschiedenen Anwendungsbereichen zum Einsatz.

1. **Einfache Warteschlange (FIFO):**
   - ▶ Tankstelle
   - ▶ Fähre
   - ▶ Pipeline
2. **Von der Restenergie der Agenten abhängige Warteschlange:**
   - ▶ Flughafen: Welches Flugzeug darf als erstes landen? Als gemeinsame Ressource dient hier der Landeplatz, und die Reihenfolge der Landungen hängt von dem Restvorrat des Treibstoffes ab.
3. **Durch mehrere Kriterien bestimmte Warteschlange**
   - ▶ Flughafen: Unterscheidet sich von dem obigen Beispiel nur in dem, dass die Reihenfolge der Landungen von mehreren Kriterien abhängig ist, wie beispielsweise:
     - Restvorrat des Treibstoffes
     - Priorität (z.B. Präsidentenflugzeug, Linienflugzeug, Privatflugzeug)
     - Flugplan
   - ▶ Bahn : Welcher Zug darf als erster die gemeinsam benutzte Strecke passieren? Als gemeinsame Ressource dienen hier die Schienen. Die Reihenfolge des Passierens ist u.a. von den folgenden Kriterien abhängig:
     - Typ des Zuges (Schnellzug, Nahverkehrszug, Güterzug, usw.)
     - Fahrplan

Die Vor– und Nachteile der beschriebenen Möglichkeiten des Aufbaus der Warteschlange werden im nächsten Abschnitt beschrieben.

5.3.2 Eigenschaften unterschiedlicher Strategien zum Aufbau von Warteschlangen

Die in Abschnitt 5.3.1 beschriebenen Möglichkeiten des Aufbaus der Warteschlange haben bestimmte Eigenschaften, die hier ausführlicher behandelt werden.

1. **Einfache Warteschlange (FIFO)**

   ▶ **Vorteile:**

   - Gerechtigkeit: Der Zutritt wird in der Reihenfolge der Requests gewährt.
   - Es sind keine Deadlocks möglich.
   - Einfachheit der Implementierung.
   - Es sind drei Nachrichten je Quellennutzung erforderlich (Request, Ok, Release).

   ▶ **Nachteile**

   - Der Wächter stellt einen möglichen Ausfallsknoten (*single point of failure*) dar.
   - Der Wächter stellt einen Flaschenhals in großen Systemen dar und führt somit zur eingeschränkten Skalierbarkeit des Algorithmus.
   - Die Agenten, die weniger Energie haben als die anderen, müssen evtl. lange warten und können deswegen an Energieverlust „sterben".

2. **Von der Restenergie der Agenten abhängige Warteschlange**

   ▶ **Vorteile:**

   - Die Agenten, die weniger Energie haben, als die anderen, werden schneller versorgt.
   - Es sind keine Deadlocks möglich.
   - Es sind drei Nachrichten je Quellennutzung erforderlich (Request, Ok, Release).

   ▶ **Nachteile**

   - Der Wächter stellt einen möglichen Ausfallsknoten (*single point of failure*) dar.
   - Der Wächter stellt einen Flaschenhals in großen Systemen dar und führt somit zur eingeschränkten Skalierbarkeit des Algorithmus.
   - Manche Agenten müssen evtl. lange warten, weil sich beim Wächter immer wieder Agenten anmelden, die kleinere Energievorräte haben.

- Die Verwaltung der Warteschlange ist komplexer, als bei der einfachen Warteschlange.

3. **Durch mehrere Kriterien bestimmte Warteschlange**

   ▶ **Vorteile:**
   Es sind alle Vorteile der von der Restenergie der Agenten abhängigen Warteschlange vorhanden. Die Flexibilität bei der Verwaltung der Warteschlange ist jedoch noch höher.

   ▶ **Nachteile:**
   Es sind alle Nachteile der von der Restenergie der Agenten abhängigen Warteschlange vorhanden. Die Warteschlangenverwaltung ist noch komplexer. Agenten mit minimalen Energievorräten werden hierbei nicht zwangsläufig zur Energiequelle durchgelassen, denn es gibt auch andere Faktoren, die evtl. berücksichtigt werden müssen, wie beispielsweise die Prioritäten der Agenten.

5.3.3 Aktionen eines Agenten beim zentralisierten Algorithmus

Wenn der Zugriff auf eine Energiequelle zentralisiert verwaltet wird, kann ein Agent eine der sechs folgenden Aktionen ausführen:

1. Einfache Bewegung
2. Bewegung zur Energiequelle
3. Umgehen eines Hindernisses
4. Warten auf die Tankerlaubnis
5. Energie tanken
6. Verlassen der Quellenumgebung
7. Sterben

Die meisten dieser Aktionen wurden schon in Abschnitt 5.1.4 ausführlich beschrieben. Deshalb wird an dieser Stelle nur die algorithmusspezifische

Aktion *Warten auf die Tankerlaubnis* erklärt.

Warten auf die Tankerlaubnis:
Wenn der Agent die Umgebung der Energiequelle erreicht hat und die Quelle besetzt ist, wird er in die Warteliste dieser Quelle eingetragen und muss solange warten, bis er die Erlaubnis zum Tanken bekommt.
Während des Wartens wird bei jedem Schritt nur ein kleiner Energieanteil abgezogen (d.h. im Leerlauf wird die Energie langsam verbraucht).

Die möglichen Verknüpfungen zwischen den einzelnen Aktionen eines Agenten beim zentralisierten Algorithmus sind in Abbildung 5.2 dargestellt.

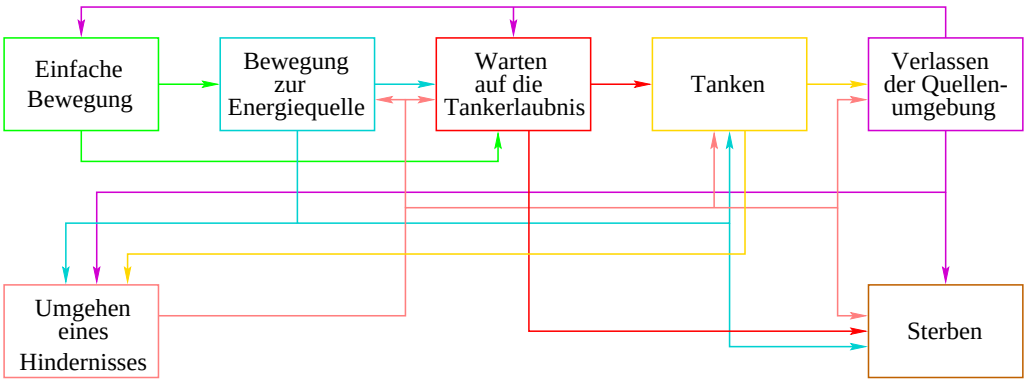


Abbildung 5.2: *Mögliche Verknüpfungen zwischen den einzelnen Aktionen eines Agenten beim zentralisierten Algorithmus*

5.4 Das Votieren

Der in den vorigen Abschnitten beschriebene zentralisierte Ansatz zur Koordination des Zugriffes auf eine gemeinsame Ressource besitzt einige Nachteile. Eine Alternative dazu stellen dezentralisierte Ansätze dar, bei denen nicht eine einzige Instanz die Entscheidungen trifft, sondern eine kollektive Entscheidung von allen Agenten getroffen wird. Eine Zwischenvariante, die Elemente von zentralisierten und dezentralisierten Ansätzen in sich vereinigt, stellt das Votieren dar.

5.4.1 Der Algorithmus

Bei dem Votier–Algorithmus existiert eine rudimentäre zentrale Instanz (ein Wächter), die das Votieren koordiniert. Das Protokoll des Votierens besteht aus der Sicht des Wächters aus drei Phasen ([12]):

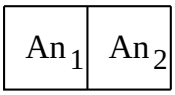
1. **Anforderungen entgegennehmen**

   Für diese Phase wird ein bestimmter Timeout festgelegt. Während dieser Zeit melden sich Agenten, die an der Energiequelle tanken wollen, beim Wächter an und schicken ihm Informationen über sich (z.B. Restenergievorrat, Priorität, usw.) zu. Agenten, die sich nach dem Timeout anmelden, werden nicht zur laufenden Votierrunde zugelassen und müssen bis zur nächsten Runde warten.

   Der Wächter verwaltet eine Liste mit allen während des Timeouts angemeldeten Agenten und ihren Parameterwerten. Sobald der Timeout abgelaufen ist, verschickt der Wächter diese Liste an alle zum Votieren zugelassenen Agenten.

2. **Votierung abwarten**

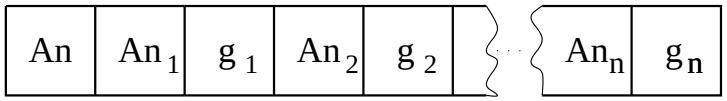
   Jeder Agent votiert für einen in der Liste enthaltenen Agenten (üblicherweise nicht für sich selbst). Diese Information schickt er an den Wächter.



An_1 – der Agent, der die Votierung abschickt
An_2 – der Agent, für den votiert wird

Abbildung 5.3: *Antwortstruktur des Agenten beim einfachen Votieren*

Anmerkung: Statt einfachem Votieren kann auch *gewichtetes Votieren* angewendet werden. D.h., ein Agent kann seine Stimme für mehrere Agenten aus der Liste abgeben, indem er denen z.B. rationale Zahlenwerte zuweist, deren Summe gleich 1 sein soll.



An – der Agent, der die Votierung abschickt
An_i – die Agenten, für die votiert wird
g_i – die Votierungen, $\sum_i g_i = 1$

Abbildung 5.4: *Antwortstruktur des Agenten beim gewichteten Votieren*

Die möglichen Antwortstrukturen eines Agenten beim Votieren sind in den Abbildungen 5.3 und 5.4 dargestellt.

3. **Entscheidung treffen**

   Nachdem der Wächter die Votierungen von allen Agenten erhält (das muss auch innerhalb eines festgelegten Zeitintervalls stattfinden), analysiert er die Ergebnisse und trifft eine Entscheidung. Diese Entscheidung wird allen Agenten mitgeteilt.

Der Agent, für den sich der Wächter entschieden hat, darf zur Quelle kommen. Während er tankt, beginnt ein neuer Timeout: Neuangekommene Agenten dürfen sich beim Wächter anmelden, die Liste wird aktualisiert, usw. Der Agent, der schon getankt hat, soll es dem Wächter mitteilen, damit er eine neue Votierrunde starten bzw. einen anderen Agenten hineinlassen kann.
Ein Agent soll sich auch dann beim Wächter abmelden, wenn er z.B. zu einer anderen Quelle gehen will oder während des Wartens an Energieverlust „gestorben“ ist.

Die Aufgabe des Wächters kann sowohl von der Energiequelle als auch von Agenten übernommen werden. Im zweiten Fall bedeutet es, dass der erste Agent, der zur Quelle gekommen ist und warten muss, zum Koordinator wird und den Timeout startet, während dessen sich andere Agenten bei ihm anmelden können. Wenn der Timeout abgelaufen ist (sinnvollerweise wenn die Quelle frei wird), nehmen alle beim Agenten–Koordinator angemeldeten Agenten am Votieren teil. Die Quelle wird während des Protokollablaufs gesperrt, damit hinzukommende Agenten nicht aus Versehen dahin gelangen

können. Sollte der Agenten–Koordinator als nächster zur Quelle kommen, übernimmt ein anderer Agent seine Aufgaben.

5.4.2 Zusätzliche Aktionen eines Agenten beim Votieren

Die meisten Aktionen, die ein Agent beim Votier–Algorithmus ausführen kann, wurden in Abschnitt 5.1.4 bereits beschrieben. Deshalb erfolgt an dieser Stelle nur die Beschreibung der Aktion *Warten* für diesen Algorithmus und der neu dazugekommenen Aktion *Votieren*, durch die die Liste der grundlegenden Aktionen eines Agenten erweitert wird.

Warten:
Wenn der Agent die Umgebung der Energiequelle erreicht hat und die Quelle besetzt ist, meldet sich der Agent bei dem Wächter dieser Quelle an, wird in die Warteliste dieser Quelle eingetragen und an den Kommunikationskanal dieser Quelle angeschlossen. Dann muss der Agent solange warten, bis die Quelle frei ist (für den Fall, dass er der einzige in der Warteliste ist) oder bis er an einer Votierung teilnehmen darf. Dementsprechend geht der Agent von dieser Aktion zu der Aktion *Energie tanken* (siehe Abschnitt 5.1.4) oder *Votieren* über.
Während des Wartens wird in jedem Zeitschritt nur ein kleiner Energieanteil abgezogen (d.h. im Leerlauf wird die Energie langsam verbraucht).

Votieren:
Wenn mehrere Agenten auf die Freigabe der Energiequelle warten, beginnt nach der Freigabe der Quelle durch den gerade tankenden Agenten eine Votierrunde. Jeder Agent, der sich vor Beginn der Votierrunde beim Wächter angemeldet hat, erhält von ihm die aktuelle Warteliste. Diese enthält als Einträge die Namen und die Verhandlungsparameter aller Agenten. Durch Vergleich dieser Parameter entscheidet sich der Agent für einen der Agenten aus der Warteliste und verschickt seine Entscheidung durch den Kommunikationskanal an den Wächter.

Die möglichen Verknüpfungen zwischen den einzelnen Aktionen beim Votier–Algorithmus sind in Abbildung 5.5 dargestellt.

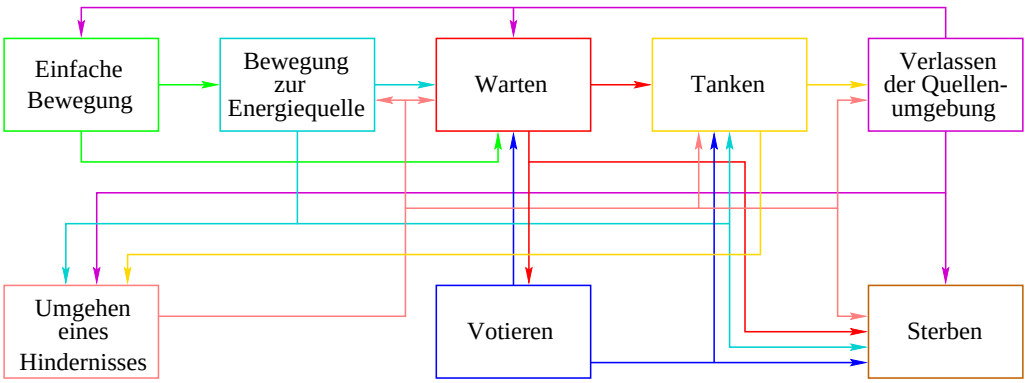


Abbildung 5.5: *Mögliche Verknüpfungen zwischen den einzelnen Aktionen eines Agenten beim Votieren*

5.4.3 Eigenschaften des Votier–Algorithmus

Die Anwendung des Votier–Algorithmus für die Koordination des Zugriffes auf gemeinsame Ressourcen hat einige Vor– und Nachteile:

- **Vorteile:**
  - Es sind keine Deadlocks möglich.
  - Die Endentscheidung des Wächters hängt von den Entscheidungen jedes einzelnen Agenten ab.
  - Es sind nur relativ wenig Nachrichten pro Votier–Runde erforderlich (Warteliste bekanntgeben, Agenten versenden ihre Votierungen)
- **Nachteile:**
  - Der Wächter bzw. der Agenten–Koordinator stellt einen möglichen Ausfallsknoten (*single point of failure*) dar.
  - Der Wächter bzw. der Agenten–Koordinator stellt einen Flaschenhals in großen Systemen dar und führt somit zur eingeschränkten Skalierbarkeit des Algorithmus.
  - Mehr Kommunikation erforderlich als beim zentralisierten Algorithmus

Zudem werden bestimmte Anforderungen an den Kommunikationskanal gestellt, wie z.B. Zuverlässigkeit, keine Duplikate, richtige Reihenfolge der Nachrichten.

5.5 Durch Agentenverhandlungen regulierter Zugriff

In den Abschnitten 5.3 und 5.4 wurden verschiedene Varianten der Koordination des Zugriffes auf eine gemeinsame Ressource vorgestellt, bei denen eine zentrale Instanz vorhanden ist.
Eine andere Möglichkeit der Koordination ergibt sich durch den Verzicht auf die zentrale Instanz. In diesem Fall wird die Koordination durch Agentenverhandlungen realisiert.

5.5.1 Algorithmus

Agenten, die an einer Quelle Energie tanken wollen, kommunizieren über einen Kanal miteinander. Über diesen Kommunikationskanal werden verschiedene Nachrichten, wie z.B. Tankanforderungen, Antworten darauf, usw. mittels *Broadcast* verschickt.
Alle zu einer Energiequelle kommenden Agenten, die an einer Verhandlung teilnehmen dürfen, sollen sich bei dem Kommunikationskanal dieser Quelle anmelden. Jeder Agent soll laufend über die Anzahl der angemeldeten Agenten informiert werden.
Wenn sich ein Agent beim Kanal abmeldet (weil er z.B. getankt hat oder zu einer anderen Energiequelle gehen will) bzw. wenn er aus dem Kanal entfernt wird (weil er während der Wartezeit an Energieverlust „gestorben“ ist), sollen alle beim Kommunikationskanal angemeldeten Agenten diese Änderung mitbekommen.

Das vorliegende Verhandlungsproblem gehört zur Problemklasse *State Oriented Domain*, die in Abschnitt 2.1.4.2 beschrieben wurde.
Um die Zugriffe auf die Energiequelle zu regulieren, können u.a. folgende **Verhandlungskriterien** benutzt werden:

1. Priorität

2. Restenergievorrat

3. Reihenfolge der Anmeldung bei der Quelle

Diese Kriterien können beliebig gewichtet sein. Im Allgemeinen können die in Abschnitt 5.3.1 beschriebenen Kriterien für den Aufbau der Warteschlange beim zentralisierten Algorithmus auch im Falle von Verhandlungen verwendet werden.

Im folgenden Abschnitt wird ein im Rahmen der vorliegenden Arbeit entwickeltes und realisiertes Verhandlungsprotokoll beschrieben. Die dabei verwendeten Abkürzungen sind in der Tabelle 5.1 zusammengefasst.

An	Agentenname
RE	Restenergievorrat eines Agenten
Pr	Agentenpriorität
Z	Zustimmung auf Tankanforderung
	$Z = \begin{cases} 0 & \text{keine Zustimmung erteilt} \\ 1 & \text{Zustimmung erteilt} \end{cases}$
ZZ	Zufallszahl

Tabelle 5.1: *Legende für Agentenprotokolle*

5.5.2 Das Verhandlungsprotokoll

Verhandlungen finden statt, falls mehrere Agenten zu einer besetzten Energiequelle kommen. Sobald die Quelle frei wird, melden sich die Agenten beim Kommunikationskanal dieser Quelle an. Danach beginnt zwischen den wartenden Agenten eine Verhandlungsrunde. Die Agenten senden eine Anfrage–Nachricht in den Kanal, die folgendermaßen aussieht:

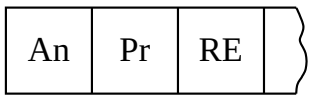


Abbildung 5.6: *Struktur einer Agentenanfrage*

Jeder Agent, der bei diesem Kommunikationskanal angemeldet ist, hört den Kommunikationskanal ab. Wenn er eine Anfrage–Nachricht bekommt, entscheidet er anhand des in Abbildung 5.7 dargestellten Entscheidungsbaums, ob er dem sendenden Agenten zustimmt oder nicht. Dabei vergleicht er die Werte seiner eigenen Verhandlungsparameter mit denen des sendenden Agenten.

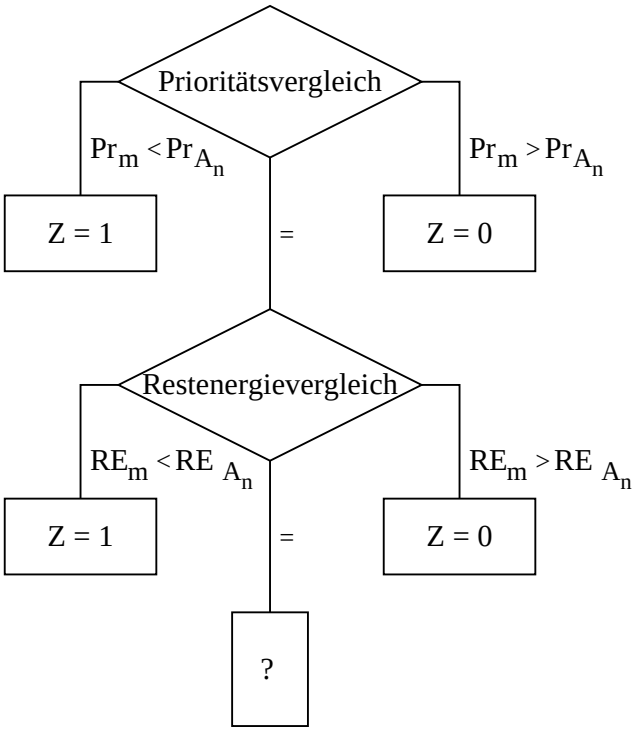


Abbildung 5.7: *Entscheidungsbaum eines an der Verhandlung beteiligten Agenten*
Die Indizes bedeuten: m – mein, A_n – sendender Agent.

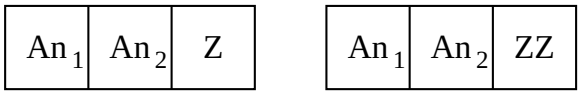
Wenn alle Parameterwerte gleich sind (siehe das in der Abb. 5.7 durch das Fragezeichen gekennzeichnete Kästchen), kann die Entscheidung aufgrund verschiedener Strategien getroffen werden:

1. Der Agent überläßt dem sendenden Agenten den Vortritt (Z=1).

2. Der Agent will selbst als erster zur Quelle kommen und verweigert deswegen dem anderen Agenten die Zustimmung.

3. Die Entscheidung wird in der zweiten Verhandlungsrunde durch den Vergleich von zwei zufällig generierten Zahlen getroffen.

Die erste beiden Strategien können zum Deadlock führen, deshalb wird im Weiteren die letzte Strategie bevorzugt.
Nach der Entscheidung sendet der Agent eine Antwort auf die Anfrage. Die Strukturen der verschiedenen Antworttypen sind in Abbildung 5.8 dargestellt. Im zweiten Fall generiert der Agent eine Zufallszahl, also einen *float*–Wert zwischen 0 und 1. Erhält der Agent, der eine Tankanforderung gesendet hat, eine solche Antwort, generiert er auch eine Zufallszahl (falls er das innerhalb der aktuellen Verhandlungsrunde noch nicht getan hat) und vergleicht die beiden Zahlen. Ist seine Zufallszahl kleiner, so wird dies als Bestätigung aufgefasst, ansonsten gilt die Anforderung als abgelehnt.

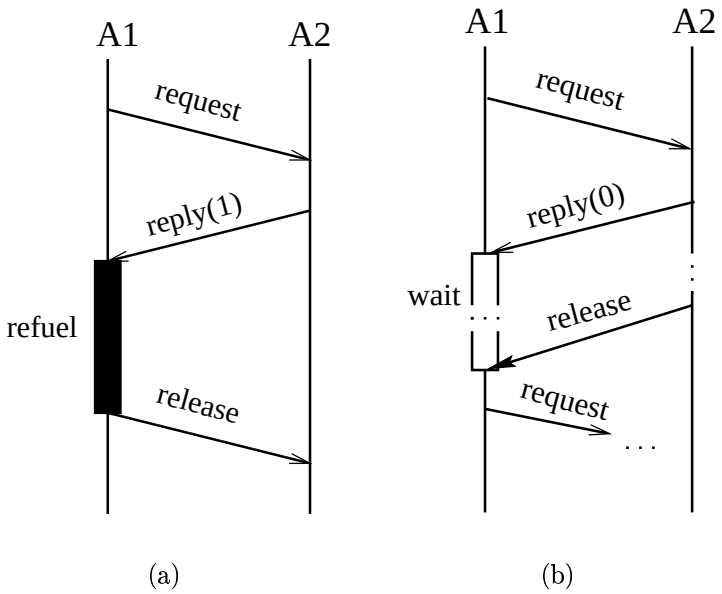


(a) (b)

An_1 – *der Agent, der eine Tank–Anforderung gesendet hat*
An_2 – *der antwortende Agent*

Abbildung 5.8: *Antwortstrukturen des Agenten bei Verhandlungen*
(a) *Die Verhandlungsparameter der Agenten unterscheiden sich*
(b) *Die Verhandlungsparameter der Agenten können keine direkte Entscheidung herbeiführen*

Zwei mögliche Verläufe des Protokolls sind in der Abbildung. 5.9 zu sehen.
Im Fall zweier Agenten bedeutet das Senden einer *release*–Nachricht, dass der andere Agent zur Quelle darf. Bei mehren Agenten beginnt beim Erhalten der *release*–Nachricht eine neue Verhandlungsrunde zwischen den restlichen Agenten.



(a) (b)

Abbildung 5.9: *Verhandlungsprotokoll*

(a) *Der Agent A1 erhält die Zustimmung auf seine Tankanforderung und tankt. Nach dem Tanken benachrichtigt er den Agenten A2, dass er fertig ist.*

(b) *Der Agent A1 erhält keine Zustimmung auf seine Tankanforderung und muss mit dem Senden einer neuen Tankanforderung warten, bis er eine release–Nachricht von dem anderen Agenten erhält.*

5.5.3 Eigenschaften des Algorithmus mit Agentenverhandlungen

Der Algorithmus zur Koordination des Zugriffes auf gemeinsame Ressourcen durch Agentenverhandlungen hat folgende Vor– und Nachteile:

- **Vorteile:**
  - Keine zentrale Instanz vorhanden
  - Jeder Agent hat ein Mitspracherecht und kann das Endergebnis der Verhandlung beeinflussen.

- **Nachteile:**
  - Deadlocks sind möglich.
  - Sehr umfangreiche Kommunikation

Zudem werden bestimmte Anforderungen an den Kommunikationskanal gestellt, wie z.B. Zuverlässigkeit, keine Duplikate, richtige Reihenfolge der Nachrichten.

5.5.4 Weitere Strategien

Bei dem durch Agentenverhandlungen regulierten Zugriff auf gemeinsame Ressourcen können die Agenten statt der bereits beschriebenen Verhandlungsstrategie auch andere Strategien benutzten. Im Folgenden sollen einige Beispiele erwähnt werden:

Hilfsstrategie: Während des Wartens (Aufenthalts in der Warteschlange) teilt ein Agent seine Restenergie mit einem anderen Agenten.

Einzelgängerstrategie: Ein Agent wird niemals einem anderen Agenten seine Zustimmung erteilen.

Täuschungsstrategie: Ein Agent gibt seinen Restenergievorrat, Priorität oder ein anderes Verhandlungskriterium nicht richtig an.

Ausserdem ist es denkbar, dass in grossen Multi–Agenten–Systemen die Agenten in Gruppen aufgeteilt werden, und die Verhandlungen zunächst innerhalb einer Gruppe und erst danach zwischen den Gruppensiegern durchgeführt werden.

Es können noch weitere, kompliziertere Varianten vorgeschlagen werden. Da aber deren Implementierung den Rahmen der vorliegenden Arbeit sprengen würde, werden sie hier nicht weiter untersucht.

5.5.5 Zusätzliche Aktionen eines Agenten bei Verhandlungen

Die in Abschnitt 5.1.4 beschriebenen Aktionen eines Agenten werden bei dem Algorithmus mit Agentenverhandlungen um weitere Aktionen erweitert.

Es kommen dabei die folgende Aktionen hinzu:

Warten:
Wenn der Agent die Umgebung der Energiequelle erreicht hat und die Quelle besetzt ist, meldet er sich bei dem Kommunakationskanal dieser Quelle an. Dann wird er in die Warteliste dieser Quelle eingetragen und muss solange warten, bis die Quelle frei ist (für den Fall, dass er der einzige in der Warteliste ist) oder bis er an einer Verhandlung zwischen den wartenden Agenten teilnehmen darf. Dementsprechend geht der Agent von dieser Aktion zu der Aktion *Energie tanken* (siehe Abschnitt 5.1.4) oder *Verhandeln* über.
Während des Wartens wird in jedem Zeitschritt nur ein kleiner Energieanteil abgezogen (d.h. im Leerlauf wird die Energie langsam verbraucht).

Verhandeln:
Wenn mehr als ein Agent auf die Freigabe der Energiequelle wartet, beginnt nach der Freigabe der Quelle durch einen getankten Agenten eine Verhandlungsrunde. Im Rahmen dieser Aktion führt ein Agent folgende Schritte aus:

- ***Tankanforderung senden:*** Der Agent sendet eine Tankanforderung in den Kommunikationskanal. Diese beinhaltet Informationen über seinen Namen und seinen restlichen Energievorrat. Die Tankanforderung erreicht durch den Kommunikationskanal alle Agenten (Broadcast), die an der Verhandlungsrunde teilnehmen.

- ***Auf Antworten warten:*** Der Agent erhält die Antworten von den anderen Agenten und analysiert sie.

- ***Entscheidung treffen:*** Bekommt der Agent von allen anderen Agenten eine positive Antwort auf seine Tankanforderung, so darf er zur Aktion *Energie tanken* übergehen. Ansonsten muss er warten.

- ***Antworten:*** Bekommt der Agent während der Verhandlungsrunde eine Tankanforderung von einem anderen Agenten, vergleicht er den Restenergievorrat des sendenden Agenten mit dem seinen und entscheidet aufgrund dieses Vergleiches, ob er dem Agenten zustimmt oder nicht. Dann sendet er eine Antwort an den anfragenden Agenten.

Die möglichen Verknüpfungen zwischen den einzelnen Aktionen eines Agenten beim Algorithmus mit Agentenverhandlungen sind in Abbildung 5.10 dargestellt.

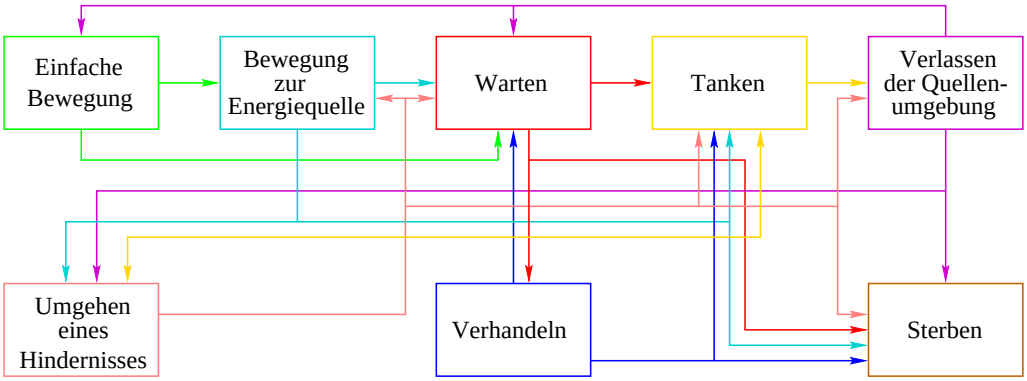


Abbildung 5.10: *Mögliche Verknüpfungen zwischen den einzelnen Aktionen eines Agenten bei Verhandlungen*

Kapitel 6

Kampf um gemeinsame Ressourcen: Implementierung

Im Rahmen dieses Teils der vorliegenden Diplomarbeit wurde das Softwarepaket zur Simulation von Agentengruppen um ein neues Modul (`package NEGOTIATION`) erweitert. Das Modul beinhaltet u.a. folgende Komponenten:

- Implementierung der drei Szenarios mit unterschiedlichen Varianten der Koordination des Zugriffes auf eine gemeinsame Ressource, nämlich
  - zentralisierte Koordination
  - Koordination durch Votieren
  - Koordination durch Verhandlungen

  (Klasse `Scenario`). Dazu gehört auch die Implementierung der Energiequellen und ihrer Umgebungen (Klassen `EnergySource, EnergySourceWithArea, VoteSourceWithArea und NegotSourceWithArea`)
- Implementierung eines einfachen abstrakten Modells eines Agenten für die obigen Szenarios (Klasse `DurstyAgent`).
- Spezialisierungen dieses Modells für die drei Varianten der Koordination des Zugriffes auf eine gemeinsame Ressource (Klassen `DurstyAgentBody, VoteAgentBody, NegotAgentBody`).
- Implementierung eines abstrakten Modells für die Aktionen eines Agenten (Klasse `Action`).

- Spezialisierungen dieses Modells für die Aktionen, die im Rahmen des implementierten Szenarios ausgeführt werden sollen (Klassen `Manoeuvre, ToFountainArea, Wait, Refuel, FromFountain, Dead, Vote und Negotiation`).
- Spezifizierung und Implementierung des Kontextes, in dem die Aktionen eines Agenten ausgeführt werden können (Interface `ActionContext` und Klasse `AgentContext`). Der Kontext wird aus Synchronisationsgründen benötigt.

6.1 Die Simulationsumgebung und die Startkonfiguration

Die Simulationsumgebung ist entsprechend der in Abschnitt 5.1.1 beschriebenen Spezifikation als eine Unterklasse von `SPACE` realisiert. Die Umgebung hat eine rechteckige Form. Die Größe der Simulationsumgebung, die Anzahl der Energiequellen und der Agenten kann beliebig gewählt werden.
Abbildung 6.1 zeigt ein Beispiel für die Startkonfiguration der Simulationsumgebung. In der Umgebung befinden sich zwei Energiequellen und sieben Agenten.

6.2 Algorithmenübergreifende Implementierung der Energiequelle

Die Energiequelle (Klasse `EnergySource`) ist als Unterklasse von `NamedBody` des Paketes `LABYRINTH` realisiert. In der vorliegenden Implementierung besitzt jede Energiequelle eine Umgebung, die von einem Agenten nur dann betreten werden darf, wenn der Agent eine Erlaubnis zum Tanken erhält. Die Umgebung der Energiequelle wurde ebenfalls als Unterklasse von `NamedBody` realisiert. Die konkrete Realisierung der Umgebung einer Energiequelle ist algorithmusabhängig und wird im weiteren für jeden Algorithmus zur Koordination des Zugriffes auf gemeinsame Ressourcen beschrieben.

Sobald ein Agent die Erlaubnis zum Tanken erhält, betritt er die Umgebung der Energiequelle. Beim Zusammenstoß mit der Quelle erhält er eine bestimmte Energiemenge (*refuelEnergy*: siehe Tab. 6.2), die er für weitere

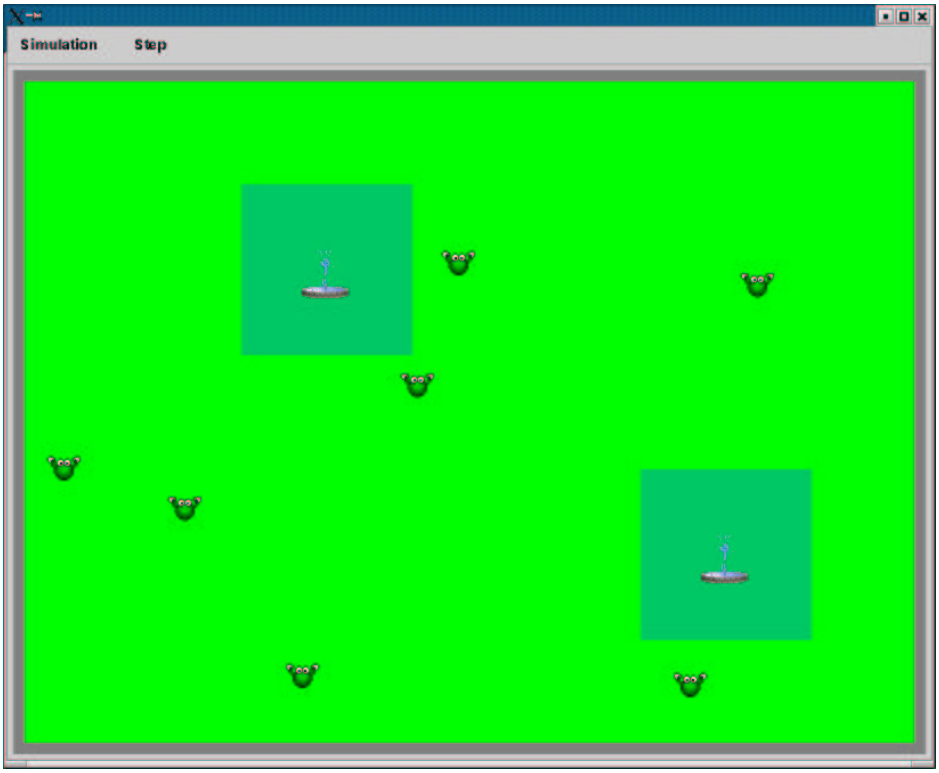


Abbildung 6.1: *Startkonfiguration der Simulation*

Aktivitäten verwenden kann. Um eine gerechte Energievergabe zu gewährleisten, erhält ein Agent während eines Tankvorganges nur einmal Energie von der Quelle, d.h. weitere Zusammenstöße, die z.B. während der Bewegung von der Energiequelle stattfinden können, bringen dem Agenten keine weiteren Energieanteile. Das geschieht auch dann nicht, wenn sein Energievorrat beim Verlassen der Quelle (Aktion *Bewegung von der Energiequelle*) kleiner als *minEnergy* wird.
Die Zustandsvariablen der Umgebung einer Energiequelle (Tab. 6.1) bilden u.a. die Grundlage für die Entscheidungen der Agenten über die unmittelbar auszuführende Aktion.

Name	Typ	Bedeutung
permitRefuel	boolean	Nimmt den Wert *true* an, wenn ein Agent tanken darf, *false* sonst.
bodyInsideArea	DurstyAgentBody (bzw. die Unterklassen davon, d.h. VoteAgentBody, NegotAgentBody)	Nimmt den Wert *null* an, wenn sich kein Agent in der Umgebung der Energiequelle befindet, zeigt sonst auf einen entsprechenden Agenten.
queue	Vector	Enthält die Liste der Agenten, die auf eine Tankerlaubnis warten und ist leer, wenn kein einziger Agent wartet.

Tabelle 6.1: *Algorithmenübergreifende Zustandsvariablen der Umgebung einer Energiequelle*

6.3 Algorithmenübergreifende Implementierung von Agenten

Ein Agent stellt eine Unterklasse von `NamedBody` aus dem Paket `LABYRINTH` dar. Diese Klasse wird dabei um einige agentenspezifische Konstanten und Variablen erweitert, die im nächsten Abschnitt beschrieben werden.

Die Agenten sind als eigenständige Threads realisiert. Während der Simulation interagieren sie miteinander und mit ihrer Umgebung. Die Kommunikation erfolgt auf der Basis von gemeinsamen Variablen, die allen Teilnehmern zugänglich sind. Nehmen mehrere Agenten bzw. Agenten und ihre Umgebung Änderungen an den gemeinsamen Variablen vor, so muss der Zugriff synchronisiert werden.

6.3.1 Parameter und Zustände eines Agenten

Die Parameter eines Agenten sind durch die in Tabelle 6.2 angegebenen Konstanten und die Zustände – durch die in Tabelle 6.3 angegebenen Zustandsvariablen realisiert.

Name	Typ	Bedeutung
initialEnergyAmount	int	anfänglicher Energievorrat des Agenten
minEnergy	int	Wenn dieser Energiewert unterschritten wird, muss der Agent sich zur Energiequelle begeben.
refuelEnergy	int	Energiemenge, die der Agent von der Quelle erhält
status	int	Priorität des Agenten

Tabelle 6.2: *Konstanten eines Agenten*

Name	Typ	Bedeutung
energyAmount	int	Der aktuelle Energievorrat eines Agenten
iAmWaiting	boolean	Nimmt den Wert *true* an, wenn der Agent auf die Tankerlaubnis bzw. auf die Votier– oder Verhandlungsrunde wartet, *false* sonst
dead	boolean	Wird auf *true* gesetzt, wenn der Agent gestorben ist, sonst ist *false*
currentAction	DurstyAgentAction	Beinhaltet die aktuelle Aktion des Agenten
myFountain	EnergySourceWithArea	Referenz auf die Energiequelle, an der getankt werden soll, *null* wenn kein Tanken erforderlich

Tabelle 6.3: *Zustandsvariablen eines Agenten*

Die Zustandsvariablen *iAmWaiting* und *dead* eines Agenten sind zu einem Kontext (Klasse `AgentContext`) zusammengefasst. Der Zugriff auf diese Kontext–Variablen soll nur synchronisiert erfolgen.

6.3.2 Aktionen eines Agenten

Die Agenten führen bestimmte Aktionen aus, die in den Abschnitten 5.1.4, 5.3.3, 5.4.2 und 5.5.5 ausführlich beschrieben wurden. Um die Möglichkeit zu haben, die Reihenfolge der ausgeführten Aktionen eines Agenten zu verfolgen, wurde bei der Implementierung für jede Aktion eine andere Agentendarstellung (Farbe) benutzt. Der Zusammenhang zwischen den Agentendarstellungen und den ausgeführten Aktionen kann der Tabelle 6.4 entnommen werden.

Die Bedingungen für die Aktionen eines Agenten, die bei allen implementierten Algorithmen ausgeführt werden können, sind in den Tabellen 6.5 und 6.6 dargestellt.
Solange der Agent noch Energie hat, überprüft er die Bedingungen der Aktionen in einer Schleife. Sollen die Bedingungen einer Aktion zutreffen, wird die entsprechende Aktion von dem Agenten ausgeführt. Daher sollen die Bedingungen der Aktionen disjunkt sein. Die Aktionen werden auf dem Agentenkontext synchronisiert ausgeführt (Abschnitt 6.4).

6.4 Behandlung von Synchronisationsproblemen

Während der Implementierung des zweiten Teils der vorliegenden Diplomarbeit sind Synchronisationsprobleme aufgetreten. Wie es in Abschnitt 6.3 beschrieben wurde, sind Agenten als eigenständige Threads realisiert worden, die miteinander und mit ihrer Umgebung durch Zugriff auf gemeinsame Variablen interagieren. Als gemeinsame Variablen dienen sowohl manche Zustandsvariablen eines Agenten (Abschnitt 6.3.1), als auch die der Umgebung einer Energiequelle (Tabelle 6.1). Wenn man die Synchronisation bei diesem Zugriff nicht beachtet, können sich daraus während der Simulationsläufe Synchronisationsprobleme ergeben, wie beispielsweise Deadlocks und falsch

Darstellung eines Agenten	Ausgeführte Aktion	Zentralisierter Algorithmus	Votier–Algorithmus	Agenten–Verhandlungen
	Einfache Bewegung	√	√	√
	Bewegung zur Energiequelle	√	√	√
	Umgehen eines Hindernisses	√	√	√
	Warten auf die Tankerlaubnis	√		
	Warten		√	√
	Votieren		√	
	Verhandeln			√
	Energie tanken	√	√	√
	Verlassen der Quellenumgebung	√	√	√
	Sterben	√	√	√

Tabelle 6.4: *Darstellung eines Agenten in Abhängigkeit von der aktuell ausgeführten Aktion.*

In den rechten Spalten der Tabelle ist angegeben, bei welchem Algorithmus die jeweilige Aktion stattfinden kann.

Auszuführende Aktion	Bedingungen
Einfache Bewegung	▶ $energyAmount > minEnergy$ ▶ Energiequelle ist nicht ausgewählt
Bewegung zur Energiequelle	▶ $0 < energyAmount \leq minEnergy$ • Energiequelle ist noch nicht ausgewählt. Das passiert beim ersten Ausführen der Aktion. oder • Energiequelle ist ausgewählt und der Agent befindet sich ausserhalb der Quellenumgebung
Umgehen eines Hindernisses	▶ der Agent hat einen Zusammenstoss und führt derzeit die Aktion *Bewegung zur Energiequelle* bzw. *Tanken* oder *Verlassen der Quellenumgebung* aus
Warten auf die Tankerlaubnis	▶ $0 < energyAmount \leq minEnergy$ ▶ Energiequelle ist ausgewählt ▶ die Quellenumgebung ist nicht leer ▶ $iAmWaiting = true$

Tabelle 6.5: *Allgemein gültige Aktionen und ihre Bedingungen (1)*

Auszuführende Aktion	Bedingungen
Energie tanken	▶ $0 < energyAmount \leq minEnergy$ ▶ Energiequelle ist ausgewählt ▶ die Erlaubnis zum Tanken ist erteilt ▶ der Agent befindet sich in der Umgebung der Energiequelle
Verlassen der Quellenumgebung	▶ Energiequelle ist ausgewählt ▶ die Erlaubnis zum Tanken ist nicht mehr gegeben ▶ der Agent befindet sich in der Umgebung der Energiequelle
Sterben	▶ $energyAmount = 0$

Tabelle 6.6: *Allgemein gültige Aktionen und ihre Bedingungen (2)*

ausgewählte Aktionen von Agenten. Die Ursache dieser Synchronisationsprobleme besteht darin, dass sowohl die Agenten selbst als auch die Umgebung unabhängig voneinander die gemeinsamen Zustandsvariablen abfragen und umsetzen.
Um den Problemen entgegenzuwirken, muss der Zugriff auf die gemeinsamen Zustandsvariablen synchronisiert werden. Wenn eine Routine aufgerufen wird, die diese Variablen möglicherweise umsetzt, kann, solange der Aufruf nicht beendet ist, keine weitere Routine, die diese Variablen verwendet, aufgerufen werden.

6.5 Implementierung des zentralisierten Algorithmus

6.5.1 Implementierung des Szenarios

Das Szenario der Simulation bei der zentralisierten Koordination des Zugriffes auf eine Energiequelle ist bereits in Abschnitt 5.3 ausführlich beschrieben worden. Wenn ein Agent weniger Energie als ein vorgegebenes Minimum (*minEnergy*, siehe Tab. 6.2) hat, dann geht zur Energiequelle, um sein Energievorrat aufzufrischen. Die Umgebung der Energiequelle dient als Wächter und verwaltet die Warteliste.

Es sind drei verschiedene Möglichkeiten des Aufbaus der Warteschlange implementiert:

- FIFO (Klasse `QueueFIFO`)
- Restenergieabhängig (Klasse `QueueEnergy`)
- Restenergie– und Prioritätsabhängig (Klasse `QueueEnergyStatus`). Dabei wurden ein bis zwei Agenten mit einer höheren Priorität versehen.

Diese drei Varianten des Aufbaus der Warteschlange und ihre Eigenschaften sind in den Abschnitten 5.3.1 und 5.3.2 ausführlich beschrieben worden.
Je nach der gewünschter Verwaltung der Warteschlange soll der Aufruf vom Programm folgendermaßen aussehen:

```
java NEGOTIATION/Scenario < FIFO | Energy | Status >
```

In einem ersten Schritt wurde die Simulationsumgebung mit nur einer Energiequelle implementiert. Um das Auswählen einer Energiequelle zu implementieren, wurde die Anzahl der Energiequellen auf zwei erhöht. Die einzige Änderung im Agentenverhalten besteht darin, dass sobald der Agent den minimalen Energievorrat erreicht, muss er sich zunächst für eine der Energiequellen entscheiden. Da das Ziel dieser Diplomarbeit in erster Linie die Entwicklung und Implementierung der Agentenkoordination beim Zugriff auf gemeinsame Ressourcen war, wurde auf die komplexeren Auswahlverfahren einer Energiequelle, die in Abschnitt 5.2 beschrieben wurden, verzichtet. Die

Entscheidung wird abhängig von der Entfernung zu den Quellen getroffen, d.h. der Agent wählt die nächst gelegene Quelle aus.

6.5.2 Implementierung der Umgebung der Energiequelle

Die Umgebung der Energiequelle ist als Instanz der Klasse `EnergySourceWithArea` implementiert.

6.5.3 Implementierung eines Agenten

Ein Agent ist als Instanz der Klasse `DurstyAgentBody` realisiert.

6.5.3.1 Zustände eines Agenten

Die Zustände eines Agenten beim zentralisierten Algorithmus des Zugriffes auf gemeinsame Ressourcen können vollständig durch die in Abschnitt 6.3.1 beschriebenen Variablen realisiert werden.

6.5.3.2 Aktionen eines Agenten

Die Aktionen eines Agenten beim zentralisierten Algorithmus des Zugriffes auf gemeinsame Ressourcen können vollständig durch die in Abschnitt 6.3.2 beschriebenen Aktionen realisiert werden.

6.5.4 Beispiel

Eine Momentanaufnahme der Simulationsumgebung ist in der Abbildung 6.2 dargestellt.

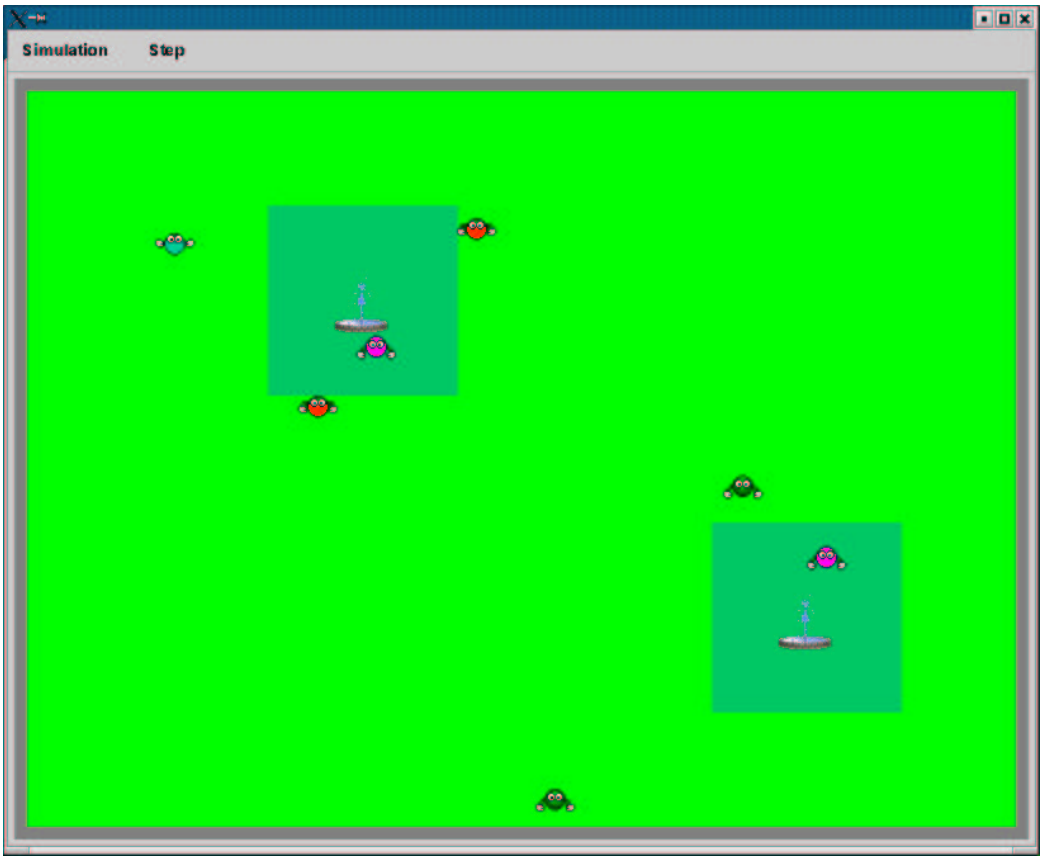


Abbildung 6.2: *Momentanaufnahme der Simulationsumgebung beim Ausführen des zentralisierten Algorithmus des Zugriffes auf gemeinsame Ressourcen*

In der Simulationsumgebung befinden sich zwei Energiequellen (Quelle1 – oben links und Quelle2 – unten rechts). Die Umgebung enthält ausserdem sieben weitere Agenten. Ein Agent hat bereits von der Quelle1 Energie erhalten und befindet sich in der Aktion *Verlassen der Quellenumgebung*. Zwei weitere Agenten warten bei der Quelle1 auf eine Tankerlaubnis. Ein anderer Agent führt die Aktion *Bewegung zur Energiequelle* aus.
Bei der Quelle2 hat ein Agent bereits Energie erhalten und verlässt die Umgebung dieser Quelle. Die zwei restlichen Agenten haben noch genug

Energievorräte und befinden sich in der Aktion *Einfache Bewegung*.

Ein Auszug der Programmausgabe, entstanden bei einem Simulationslauf der Programmvariante mit FIFO–Warteschlangenaufbau, ist in der Abbildung 6.3 dargestellt.

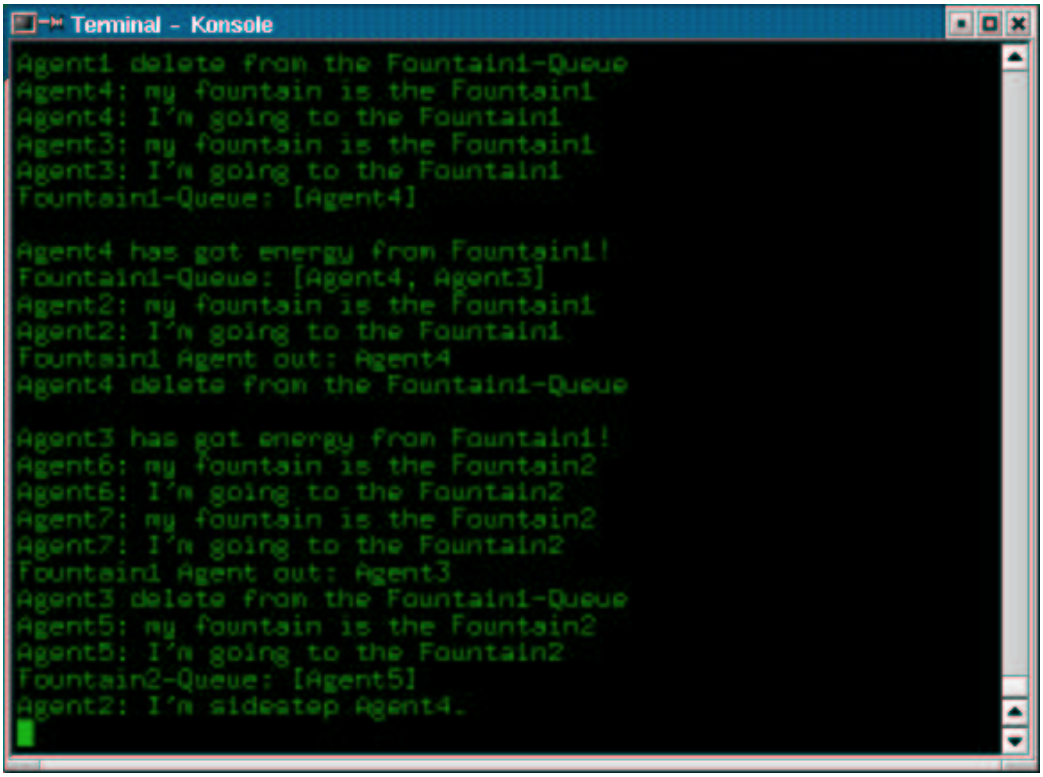


Abbildung 6.3: *Programmausgabe während der Ausführung des zentralisierten Algorithmus des Zugriffes auf gemeinsame Ressourcen (FIFO)*

Anhand dieser Ausgabe kann man den Aufbau der Warteschlange und den Simulationsablauf verfolgen.

6.6 Implementierung des Votier–Algorithmus

6.6.1 Implementierung des Szenarios

Der Votier–Algorithmus wurde in Abschnitt 5.4.1 beschrieben. Der Aufruf der Programmvariante mit dem Votier–Algorithmus sieht folgendermaßen aus:

```
java NEGOTIATION/Scenario Vote
```

Jede Energiequelle besitzt einen Wächter (ihre Umgebung), der die Zugriffe auf die Quelle koordiniert und einen Kommunikationskanal, durch den Nachrichten zwischen dem Wächter und den Agenten ausgetauscht werden können.

Wenn ein Agent weniger Energie als ein vorgegebenes Minimum (*minEnergy*, siehe Tab. 6.2) hat, dann wählt er eine Energiequelle aus und geht in die Richtung dieser Quelle, um seinen Energievorrat aufzufrischen. Wenn der Agent zur Energiequelle kommt, und die Quelle frei ist, darf der Agent sofort tanken. Ist die Quelle von einem anderen Agenten bereits besetzt worden, so muss der neu hinzugekommene Agent warten. Dabei meldet er sich bei dem Wächter der Energiequelle an. Falls sich während der Zeit, in der die Energiequelle besetzt ist, mehrere Agenten bei der Quelle versammeln, dann beginnt nach der Freigabe der Quelle eine Votierrunde zwischen den wartenden Agenten. Die Votierrunde gilt an dieser Stelle als abgeschlossen, so dass Agenten, die während der Votierung zur Quelle kommen, an dieser Votierung nicht mehr teilnehmen können. Sie müssen also bis zur nächsten Votierrunde warten.

Jeder Agent, der an der Votierrunde teilnimmt, wird an den Kommunikationskanal dieser Quelle angeschlossen und erhält vom Wächter eine aktuelle Warteliste, die Einträge bestehend aus den Agentennamen und ihren aktuellen Verhandlungsparametern enthält. Der Agent analysiert die Warteliste und entscheidet sich für einen der in der Liste enthaltenen Agenten. Zur Zeit erfolgt diese Entscheidung anhand des Vergleichs der Energierestvorräten der Agenten, d.h. der Agent mit dem kleinsten Energievorrat wird bevorzugt.

Dabei votiert ein Agent immer für den ersten in der Warteliste enthaltenen und über den kleinsten Energierestvorrat verfügenden Agenten. Sollten also mehrere Agenten den gleichen Restvorrat an Energie haben, wird der erste zur Energiequelle gekommene Agent bevorzugt. Damit werden zwei Verhandlungsparameter gleichzeitig berücksichtigt: der Energierestvorrat und der Zeitpunkt der Anmeldung bei der Quelle.

Der Agent gibt seine Votierung über den Kanal bekannt. Eine Votierung ist als Instanz der Klasse `VoteEvent` realisiert und hat die Struktur, die in der Abbildung 5.3 dargestellt ist. Wenn der Wächter die Votierungen von allen Agenten erhalten hat, trifft er eine Entscheidung, d.h. er wählt den Agenten aus, der die Mehrheit der Stimmen bekommen hat. Der ausgewählte Agent erhält die Erlaubnis zum Tanken.

Während der Agent tankt, werden alle nach dem Beginn der letzten Votierrunde zur Quelle gekommenen Agenten zu einer neuen Runde zugelassen. Sie müssen aber mit dem Votieren warten, bis die Quelle freigegeben wird. Wenn der Agent die Quelle verlässt, meldet er sich beim Wächter ab.

Beim Start der Simulation wird für jede Energiequelle ein Kanalfenster geöffnet. In diesem Fenster kann man die Votierungen der einzelnen Agenten beobachten.

6.6.2 Implementierung der Umgebung der Energiequelle

Die Umgebung der Energiequelle ist als Instanz der Klasse `VoteSourceWithArea` implementiert. Sie dient als Wächter bei der Koordination des Zugriffes auf die Energiequelle. Die zusätzlichen Zustandsvariablen der Umgebung sind in der Tabelle 6.7 dargestellt.

6.6.3 Implementierung eines Agenten

Ein Agent ist als Instanz der Klasse `VoteAgentBody` realisiert.

Name	Typ	Bedeutung
queueSnapshot	Vector	Enthält eine Kopie der *queue* für die aktuelle Votierrunde: eingetragen sind nur die Agenten, die an der Votierrunde teilnehmen dürfen.
votings	Hashtable	Ist *leer*, wenn keine Votierrunde stattfindet, enthält ansonsten die Namen der Agenten, die schon votiert haben.

Tabelle 6.7: *Zusätzliche Zustandsvariablen der Umgebung einer Energiequelle beim Votier–Algorithmus*

6.6.3.1 Zustand eines Agenten

Die Zustände eines Agenten beim Votier–Algorithmus für den Zugriff auf gemeinsame Ressourcen können durch die Zustandsvariablen, die in Tabelle 6.3 vorgestellt wurden, beschrieben werden.

6.6.3.2 Aktionen eines Agenten

Die Bedingungen für die zusätzliche Aktion eines Agenten beim Votier–Algorithmus sind in der Tabelle 6.8 dargestellt.

Auszuführende Aktion	Bedingungen
Votieren	▶ Energiequelle ist ausgewählt ▶ $0 < energyAmount \leq minEnergy$ ▶ $iAmWaiting = true$ ▶ eine Votierrunde findet statt ▶ der Agent darf an dieser Votierrunde teilnehmen

Tabelle 6.8: *Zusätzliche Aktion eines Agenten und ihre Bedingungen beim Votier–Algorithmus*

6.6.4 Beispiel

Ein Auszug der Programmausgabe (Abb. 6.4), die während der Simulation des Votier–Algorithmus beim konkurrierenden Zugriff auf eine Energiequelle in einem Kanalfenster zu sehen ist, spiegelt den Nachrichtenaustausch zwischen den Agenten und dem Wächter in dem Kommunikationskanal wieder.

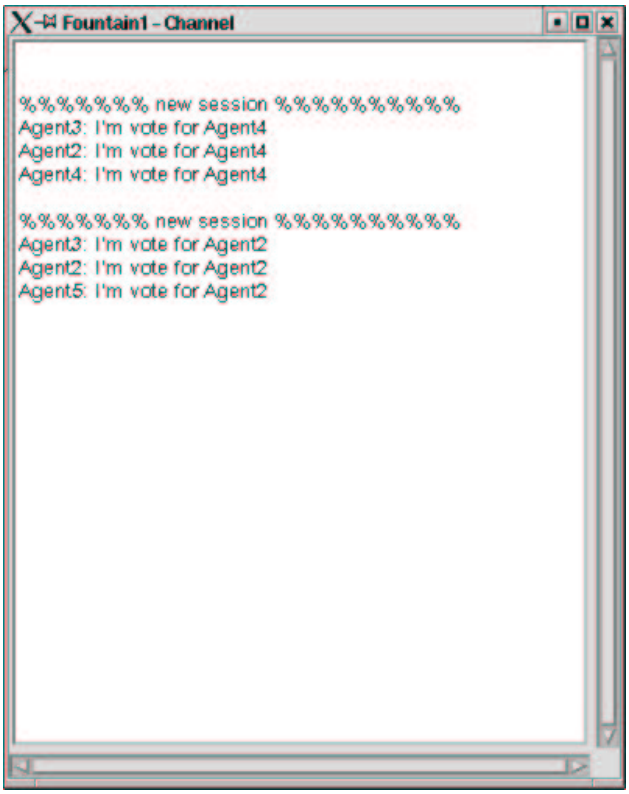


Abbildung 6.4: *Ausgabe in einem Kanalfenster beim Votieren*

In der Abbildung sind die Votierungen von Agenten innerhalb von zwei Votierrunden dargestellt. An der ersten Votierrunde nehmen drei Agenten (Agent2, Agent3 und Agent4) teil. Alle drei Agenten versenden in den Kanal eine Votierung für Agent4. Nach dem Erhalt dieser Votierungen muss sich der Wächter für Agent4 entscheiden, d.h. Agent4 erhält die Erlaubnis zum Tanken.

Während der ersten Votierrunde ist Agent5 hinzugekommen. Deshalb beginnt eine neue Votierrunde zwischen den drei wartenden Agenten, sobald Agent4 die Energiequelle freigibt. Laut den in den Kommunikationskanal gesendeten Votierungen der Agenten, votieren alle drei Agenten für Agent2. Deshalb erhält Agent2 die Erlaubnis zum Tanken vom Wächter, nachdem er alle Votierungen erhalten und analysiert hat.

6.7 Implementierung der Agentenverhandlungen

6.7.1 Implementierung des Szenarios

Der Algorithmus zur Koordination des Zugriffes auf gemeinsame Ressourcen durch Agentenverhandlungen wurde in Abschnitt 5.5.1 beschrieben. Wenn die Programmvariante mit Agentenverhandlungen aufgerufen werden soll, sieht der Aufruf folgendermaßen aus:

```
java NEGOTIATION/Scenario Negotiation
```

Jede Energiequelle besitzt einen Kommunikationskanal, in dem die Verhandlungen zwischen den Agenten stattfinden. Die Nachrichten, die über diesen Kanal gesendet werden, sind als Instanzen der Klassen `NegEvent`, `AnswerEvent` und `RandomEvent` implementiert. Die Strukturen dieser Nachrichten sind in den Abbildungen 5.6 und 5.8 dargestellt.

Wenn ein Agent weniger Energie als ein vorgegebenes Minimum (*minEnergy*, siehe Tab. 6.2) hat, dann wählt er eine Energiequelle aus und geht in Richtung dieser Quelle, um seinen Energievorrat aufzufrischen. Wenn der Agent zur Energiequelle kommt, und die Quelle frei ist, darf der Agent sofort tanken. Ist die Quelle bereits von einem anderen Agenten besetzt worden, so muss der neu hinzugekommene Agent warten. Falls sich während der Zeit, in der die Energiequelle besetzt ist, mehrere Agenten bei der Quelle versammeln, dann beginnt nach der Freigabe dieser Quelle eine Verhandlungsrunde zwischen den wartenden Agenten. Nach Beginn der Verhandlungen werden keine weiteren Agenten mehr zu den Verhandlungen zugelassen, d.h. sie müssen bis zur nächsten Verhandlungsrunde warten.

Die Agenten, die an der Verhandlungsrunde teilnehmen, melden sich bei dem Kanal der Energiequelle an. Während der Verhandlung sendet jeder Agent eine Tankanforderung über den Kanal. Diese Tankanforderung enthält den Agentennamen und den Restenergievorrat des Agenten. Sie wird mittels *Broadcast* an alle an der aktuellen Verhandlungsrunde teilnehmenden Agenten verschickt.

Jeder Agent, der eine Tankanforderung bekommt, vergleicht seinen Restenergievorrat mit dem des sendenden Agenten. Falls der eigene Restenergievorrat kleiner ist, wird ein „Nein“, anderenfalls ein „Ja“ an den Agenten zurückgeschickt. Sollen zwei Agenten gleichgroße Vorräte besitzen, so wird von jedem Agenten eine Zufallszahl, deren Wert im Intervall zwischen 0 und 1 liegt, generiert und an den anderen Agenten verschickt. Die Entscheidung wird dann anhand des Vergleiches der Zufallszahlen getroffen.

Jeder an der Verhandlung teilnehmende Agent zählt die Antworten auf seine Tankanforderung. Ist die Anzahl der positiven Antworten um eins kleiner als die Anzahl der an der Verhandlungsrunde teilnehmenden Agenten (d.h. alle andere Agenten haben mit „Ja“ geantwortet), darf der Agent zur Quelle gehen und tanken. Während er tankt, werden alle wartenden Agenten zu einer neuen Verhandlungsrunde zugelassen. Sie müssen aber mit den Verhandlungen warten bis die Quelle freigegeben wird.
Wenn der Agent die Quelle verlässt, meldet er sich bei dem Kanal ab.

Beim Start der Simulation wird für den Kommunikationskanal der Energiequelle ein Kanalfenster geöffnet. In diesem Fenster kann man die Verhandlungen zwischen den Agenten beobachten: es werden Tankanforderungen und die Antworten der Agenten darauf ausgegeben.

6.7.2 Implementierung der Umgebung der Energiequelle

Die Umgebung der Energiequelle ist als Instanz der Klasse `NegotSourceWithArea` implementiert. Die zusätzliche Zustandsvariable der Umgebung ist in der Tabelle 6.9 beschrieben.

Name	Typ	Bedeutung
queueSnapshot	Vector	Enthält eine Kopie der *queue* für die aktuelle Verhandlungsrunde: es sind nur die Agenten eingetragen, die an der Verhandlungsrunde teilnehmen dürfen.

Tabelle 6.9: *Zusätzliche Zustandsvariable der Umgebung einer Energiequelle beim Algorithmus mit Agentenverhandlungen*

6.7.3 Implementierung eines Agenten

Ein Agent ist als Instanz der Klasse `NegAgentBody` realisiert.

6.7.3.1 Zustand eines Agenten

Die Zustände eines Agenten bei der Koordination des Zugriffes auf gemeinsame Ressourcen durch Agentenverhandlungen können durch die Zustandsvariablen, die in der Tabelle 6.3 vorgestellt wurden, und die zusätzliche Zustandsvariablen aus Tabelle 6.10 beschrieben werden.

Name	Typ	Bedeutung
negCounter	int	Anzahl der Agenten, die auf die Tankanforderung des Agenten positiv geantwortet haben
negEvent	NegEvent	Ist *null*, wenn der Agent während der aktuellen Verhandlungsrunde noch keine Tankanforderung gesendet hat.

Tabelle 6.10: *Zusätzliche Zustandsvariablen eines Agenten beim Algorithmus mit Agentenverhandlungen*

6.7.3.2 Aktionen eines Agenten

Die Bedingungen für die zusätzliche Aktion eines Agenten beim Algorithmus mit Agentenverhandlungen sind in der Tabelle 6.11 dargestellt.

Auszuführende Aktion	Bedingungen
Verhandeln	▶ Energiequelle ist ausgewählt ▶ $0 < energyAmount \leq minEnergy$ ▶ $iAmWaiting = true$ ▶ eine Verhandlungsrunde findet statt ▶ der Agent darf an dieser Verhandlungsrunde teilnehmen

Tabelle 6.11: *Zusätzliche Aktion eines Agenten und ihre Bedingungen beim Algorithmus mit Agentenverhandlungen*

6.7.4 Beispiel

Ein Auszug der Programmausgabe (Abb. 6.5), die während der Simulation von Agentenverhandlungen beim konkurrierenden Zugriff auf eine Energiequelle in einen Kanalfenster erfolgt ist, spiegelt den Nachrichtenaustausch zwischen den Agenten in dem Kommunikationskanal wieder.
In der Abbildung ist der Nachrichtenaustausch zwischen verschiedenen Agenten während zwei Verhandlungsrunden dargestellt. An der ersten Verhandlungsrunde nehmen zwei Agenten (Agent2 und Agent4) teil. Laut den Tankanforderungen, die sie versendet haben, hat Agent4 einen kleineren Energievorrat als Agent2. Agent2 stimmt der Tankanforderung von Agent4 zu, und somit kann Agent4 tanken gehen.
Während der ersten Verhandlungsrunde sind noch zwei weitere Agenten (Agent3 und Agent5) hinzugekommen. Sobald Agent4 die Energiequelle freigibt, beginnt eine neue Verhandlungsrunde zwischen den drei wartenden Agenten. Laut den in den Kommunikationskanal gesendeten Tankanforderungen der Agenten, verfügt Agent2 jetzt über den minimalen Energievorrat. Der Energievorrat von Agent5 ist am grössten, deshalb erhält der Agent keine Zustimmung auf seine Tankanforderung von den beiden anderen Agenten. Agent3 erhält nur die Zustimmung von Agent5. Agent2 bekommt aber die Zustimmung von den beiden anderen Agenten und somit die Erlaubnis zum Tanken.

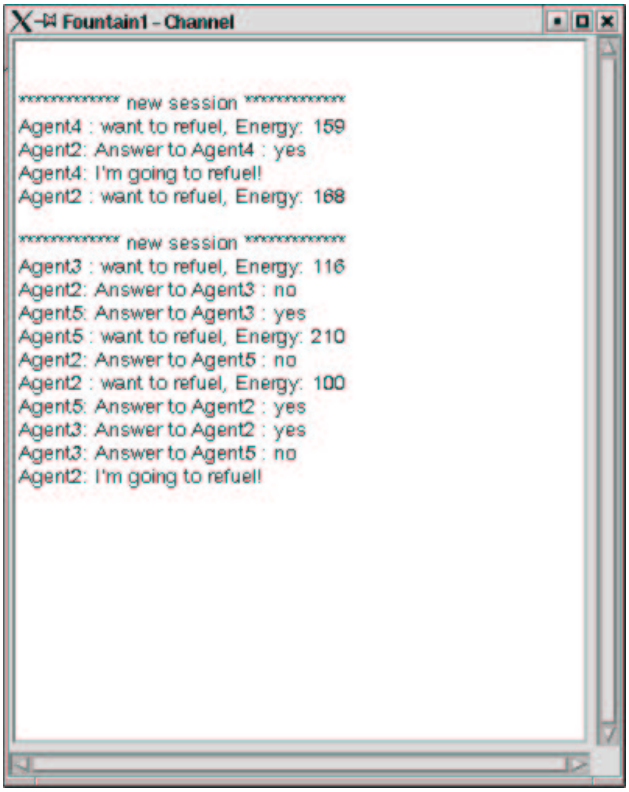


Abbildung 6.5: *Ausgabe in einem Kanalfenster bei Agentenverhandlungen*

Kapitel 7

Zusammenfassung und Ausblick

7.1 Ergebnisse

Im Rahmen der vorliegenden Diplomarbeit wurde ein bereits bestehendes Softwarepaket zur Simulation von Agentengruppen um zwei neue Module erweitert. Zum einen, wurde dabei ein Modul entwickelt und implementiert, das eine verteilte Simulation ermöglicht. Die Interprozesskommunikation, wie beispielsweise die Kommunikation zwischen den Agenten, erfolgt dabei auf Basis der CORBA–Architektur übers Netz. Es ist auch gelungen, eine transparente Integration der neu entwickelten Komponenten in das existierende Konzept zu erreichen. Es wurde experimentell festgestellt, dass sich der Einsatz dieser Komponente bei der jetzigen Komplexitätsstufe einzelner Agenten ab ca. 10 Agenten empfiehlt. Unterhalb dieser Schwelle können Agenten noch auf einer Maschine als eigenständige Threads simuliert werden. Es läßt sich vermuten, dass bei wachsender Komplexität einzelner Agenten diese Schwelle weiter sinkt.
Im zweiten Teil der Arbeit wurden verschiedene Algorithmen zur Koordination des Zugriffes auf gemeinsame Ressourcen realisiert. Dabei wurde eine einfache Agenten–Architektur entworfen, implementiert und getestet. Es wurde sowohl ihre prinzipielle Eignung für die Simulationszwecke gezeigt, als auch inwiefern und in welcher Richtung sie weiter entwickelt werden soll. Einige Erweiterungsmöglichkeiten sind im nächsten Abschnitt zusammengefasst.

7.2 Mögliche Erweiterungen

Die vorliegende Arbeit kann an mehreren Stellen fortgesetzt werden. Dazu zählen u.a. folgende Punkte:

- Im Rahmen der vorliegenden Arbeit wurden externe Aufgaben eines Agenten nicht simuliert, bzw. sie waren in der Aktion *Einfache Bewegung* zusammengefasst.

  Sobald die damit verbundenen Aktivitäten der Agenten simuliert werden, wächst die Anzahl der Aktionen, die ein Agent ausführen kann.

  In diesem Fall wird es schwierig sein, die Disjunktheit der Bedingungen der einzelnen Aktionen zu erhalten. Eine mögliche Lösung dieses Problems besteht in der Erweiterung des bestehenden Agentenmodells. Dabei soll das Agentenmodell um eine weitere Komponente (Entscheidungseinheit) erweitert werden, die es einem Agenten ermöglicht, aus der gesamten Liste aller Aktionen zunächst eine Menge der in Frage kommenden Aktionen auszuwählen und danach anhand bestimmter Kriterien die endgültige Entscheidung zu treffen.

- Die zur Zeit realisierte Strategie des Umgehens eines Hindernisses scheint nicht optimal zu sein. Besonders problematisch ist dies beim Verlassen der Umgebung der Energiequelle: Ein zurückkehrender Agent verbraucht gelegentlich viel Energie dadurch, dass er ständig mit anderen Agenten zusammenstößt, bis er lediglich einen freien Ausgang finden kann. Ihrerseits verlieren die an der Quelle wartenden Agenten viel Energie bis die Quelle frei ist. Diese Tendenz steigt mit der Erhöhung der Anzahl der Agenten und (oder) der Verminderung der Anzahl der Energiequellen.

- Man kann versuchen, die komplizierteren Möglichkeiten der Auswahl einer Energiequelle bzw. das Erforschen der Lage der Quellen, die in Abschnitt 5.2 beschrieben wurden, zu realisieren und zu vergleichen.

- Z.Z. erhält jeder Agent beim Start der Simulation die gleichen Werte von *minEnergy* und *refuelEnergy*. Man kann Agenten mit unterschiedlichen Einstellungen dieser Parameter simulieren, um festzustellen, welche Einstellungen in einer gegebenen Simulationsumgebung zu besseren Überlebenschancen der Agenten führen.

- Bezogen auf den Zugriff auf gemeinsame Ressourcen, kann man die Übernahme der Koordinator–Aufgaben durch einen Agenten implementieren. Dies hat aber mit grosser Wahrscheinlichkeit keine Auswirkung auf das Verhalten des Systems. Wesentlich interessanter wäre es, andere, wie beispielsweise die in Abschnitt 5.5.4 beschriebenen Verhandlungsstrategien, zu implementieren und zu vergleichen.

Literaturverzeichnis

[1] V. Avrutin, R. Lammert, D. Lippold, P. Levi, and M. Schanz. Rechnergestützte Simulation autonomer Agenten in einer sozionischer Welt. *SozionikAktuell*, 2001.

[2] IONA Technologies, Inc. *Orbix Homepage, URL: http://www.iona.com/docs/manuals/orbix2000/*, 2000.

[3] IONA Technologies, Inc. *ORBacus for C++ und Java*, Version 4.0.5 edition, 2001.

[4] IONA Technologies, Inc. *ORBacus Homepage, URL: http://www.orbacus.com*, 2001.

[5] Darko Ivancˆan. Erweiterung der Teamfähigkeit für ein Team fußballspielender Roboter. Diplomarbeit Nr.1866, Universität Stuttgart, 2000.

[6] Guido Krüger. *Java 1.1 lernen.* Addison-Wesley, Bonn, 1997.

[7] Paul Levi. Architectures of individual and distributed autonomous agents. In *Proceedings of the 2nd International Conference on Intelligent Autonomous Systems*, 1989.

[8] Object Management Group, Inc. *URL: http://www.omg.org*, 1997-2002.

[9] Object Oriented Concepts, Inc. *CORBA/C++ Programming with ORBacus*, December 2000.

[10] Jens-Peter Redlich. *Corba 2.0. Praktische Einführung für C++ und Java.* Addison-Wesley, 1996.

[11] Jeffrey S. Rosenschein and Gilad Zlotkin. *Rules of Encounter: Designer Conventions for Automated Negotiation among Computer.* MIT Press, 1994.

[12] Kurt Rothermel. Skript zur Vorlesung Grundlagen der Verteilten Systeme. Universität Stuttgart, SS 1997.

[13] Michael Schanz. Skript zur Vorlesung Einführung in die Verteilte KI. Universität Stuttgart, WS 2000.

[14] Peter Schmutter. Eine Einführung in Multiagentensysteme und deren Verwirklichung im Roboterfußball. URL: http://www.schmutter.de/deutsch/myactivities/computerscience/masreferat/page.html, April 2000.

[15] Sun Microsystems. *The Java Tutorial. Java 2 Platform, URL: http://java.sun.com/products/jdk/1.2/docs/api/index.html*, Version 1.2.2. edition, 2001.

[16] Gerhard Weiß, editor. *Multiagent Systems.* MIT Press, 1999.

[17] Michael Wurst. Untersuchung und Entwicklung von Verfahren zum kooperativen Maschinellen Lernen in Multi-Agenten Systemen. Diplomarbeit Nr.1935, Universität Stuttgart, 2001.