

**Prüfer:** Prof. Dr. rer. nat. Kurt Rothermel

**Betreuer:** Dipl.-Inform. Markus Straßer

**Beginn am:** 15. April 1999

**Beendet am:** 14. Oktober 1999

**CR-Nummer:** C.2.2, C.2.4, D.1.5, D.4.5, H.2.4

Diplomarbeit Nr. 1770

**" Fehlertoleranz in Mole "**

Konstantinos Papoulidis

Fakultät Informatik  
Institut für Parallele und Verteilte  
Höchstleistungsrechner  
Universität Stuttgart  
Breitwiesenstr. 20-22  
70565 Stuttgart

DIPLOMARBEIT

---

# **Fehlertoleranz in Mole**

---

---

## Inhaltsverzeichnis

<b>Inhaltsverzeichnis .....</b>	<b>I</b>
<b>Abbildungsverzeichnis .....</b>	<b>IV</b>
<b>Kapitel 1 : Einleitung .....</b>	<b>1</b>
1.1 Einführung .....	1
1.2 Aufgabenstellung .....	2
1.3 Überblick über die Diplomarbeit .....	3
<b>Kapitel 2 : Agentensystem Mole 3.0 .....</b>	<b>4</b>
2.1 Agenten .....	5
2.1.1 Benutzer-Agenten .....	5
2.1.2 System-Agenten .....	5
2.2 Kommunikationsmechanismen .....	6
2.2.1 Messages .....	6
2.2.2 Remote Procedure Call .....	6
2.2.3 Session .....	7
2.2.4 Lokale / Globale Kommunikation .....	8
2.3 Location .....	8
2.4 Engine .....	9
<b>Kapitel 3 : Exactly-Once-Protokoll .....</b>	<b>10</b>
3.1 Einführung .....	10
3.2 Beschreibung des Exactly-Once-Protokolls .....	11
3.3 Anmerkungen .....	14

<b>Kapitel 4 : Mole API .....</b>	<b>16</b>
4.1 Kommunikation .....	16
4.1.1 Partner .....	16
4.1.2 Problem .....	16
4.1.3 Einschränkungen .....	17
4.2 Agentenerzeugung .....	19
4.3 Migration .....	20
4.3.1 Bisherige Migration .....	20
4.3.2 Anwendung der Reiseroute .....	20
4.4 Entwurf der Mole API .....	21
<b>Kapitel 5 : Stage Construction .....</b>	<b>31</b>
5.1 Vorbemerkungen .....	31
5.2 Interaktion zwischen Protokoll und Reiseroute .....	31
5.3 Anzahl der Stage - Locations .....	32
5.4 Typen von Stage - Locations .....	32
5.5 Performance Betrachtungen .....	33
5.6 Beschreibung der Stage - Construction .....	34
5.7 Entwurf der Stage - Construction .....	43
5.8 Anforderungen an die Reiseroutenspezifikation .....	46
<b>Kapitel 6 : Integration des Exactly - Once - Protokolls in MOLE .....</b>	<b>47</b>
6.1 Vorbemerkungen .....	47
6.2 Möglichkeiten der Integration .....	47
6.3 Entwurf .....	48

<b>Kapitel 7 : Zusammenfassung und Ausblick .....</b>	<b>50</b>
7.1 Zusammenfassung .....	50
7.2 Ausblick .....	50
<b>Anhang "A" .....</b>	<b>52</b>
Anhang "A.1" : Kapitel 4 .....	52
Anhang "A.2" : Kapitel 5 .....	74
<b>Anhang "B" : Projektplan für die Diplomarbeit .....</b>	<b>78</b>
<b>Anhang "C" : Literaturverzeichnis .....</b>	<b>83</b>

---

## Abbildungsverzeichnis

Abbildung 2-1: Systemmodell von "Mole" .....	4
Abbildung 2-2: Kommunikationsmechanismus "Messages" .....	6
Abbildung 2-3: Kommunikationsmechanismus "RPC" .....	7
Abbildung 2-4: Kommunikationsmechanismus "Session" .....	7
Abbildung 3-1: Transaktionale Message Queues .....	12
Abbildung 3-2: Agentenausführung in 2 Stufen .....	13
Abbildung 4-1: Kommunikationspartner in Mole 3.0 .....	16
Abbildung 4-2: Kommunikationspartner nach dem Protokoll .....	17
Abbildung 4-3: "RPC" Kommunikation .....	18
Abbildung 4-4: Session - RPC .....	19
Abbildung 4-5: Eingeschränkte Mole - API (Teil 1) .....	22
Abbildung 4-6: Eingeschränkte Mole - API (Teil 2) .....	23
Abbildung 4-7a: Agent - Klasse .....	23
Abbildung 4-7b: Agent - Klasse .....	24
Abbildung 4-8a: Location - Klasse .....	25
Abbildung 4-8b: Location - Klasse .....	26
Abbildung 4-9: Session - Klasse .....	27
Abbildung 4-10: Message - Klasse .....	27
Abbildung 4-11: MoleprotWrapper - Klasse .....	28
Abbildung 4-12: RemoteLocation - Schnittstelle .....	28
Abbildung 4-13: IllegalRPCException - Klasse .....	28

---

Abbildung 4-14: TransactionContextException - Klasse .....	28
Abbildung 4-15: MoleThread - Klasse .....	29
Abbildung 4-16: IllegalThreadCreateException - Klasse .....	29
Abbildung 4-17: MCP - Klasse .....	29
Abbildung 4-18: StandardAgentThread - Klasse .....	30
Abbildung 5-1: Stage Construction und Reiseroute .....	32
Abbildung 5-2: Reiserouten - Spezifikation .....	34
Abbildung 5-3a: "result - Vektor" .....	35
Abbildung 5-3b: "result - Vektor" .....	36
Abbildung 5-3c: "result - Vektor" .....	37
Abbildung 5-3d: "result - Vektor" .....	38
Abbildung 5-3e: "result - Vektor" und "total - Vektor" .....	38
Abbildung 5-3f: "result - Vektor" und "total - Vektor" .....	39
Abbildung 5-3g: "total - Vektor" .....	39
Abbildung 5-4: NextW .....	40
Abbildung 5-5: Fall 2a .....	41
Abbildung 5-6: Fall 2b .....	41
Abbildung 5-7: Fall 3 .....	42
Abbildung 5-8: Überblick des Entwurfs Stage - Construction .....	43
Abbildung 5-9: Agent - Klasse .....	43
Abbildung 5-10: Location - Klasse .....	44
Abbildung 5-11: Engine - Klasse .....	45
Abbildung 6-1: Verarbeitungsmodell .....	47

# 1 Einleitung

## 1.1 Einführung

Unter Mobilien Agenten versteht man ein neues Programmiermodell, welches im Bereich der Verteilten Systeme angewendet wird. Mobile Agenten besitzen die Fähigkeit, sich autonom in einem Netzwerk von Rechnern (die Rechner werden im folgenden als Knoten bezeichnet) zu bewegen und gezielt eine Aufgabe zu erledigen. Sie können also von Knoten, welche als "Orte" abstrahiert werden, zu Knoten migrieren und mit den an diesem Ort vorhandenen Ressourcen, welche durch spezielle Agenten repräsentiert werden, kommunizieren und angebotene Dienste in Anspruch nehmen.

Auf beiden Seiten (Knoten) muß ein "Agentensystem" installiert sein, das eine Umgebung für die Ausführung eines Agenten zur Verfügung stellt und eine Abstraktion des darunterliegenden Netzwerkes als auch des Betriebssystems repräsentiert. Wenn bei der Migration die Agenten sowohl ihren Ausführungszustand als auch den Zustand der Daten behalten, wird dies als "Strong Migration" bezeichnet. Im Gegensatz dazu, wird bei "Weak Migration" nur der Code und die Daten des Agenten, aber ohne Ausführungszustand, übertragen. Nachdem die Ausführung des Agenten auf dem aktuellen Knoten gestoppt wird, der Agent entfernt und übertragen wird auf einen anderen Knoten, wird die Ausführung des Agenten dort fortgesetzt. Zur Bearbeitung einer Aufgabe können Mobile Agenten mit anderen Agenten über verschiedene Mechanismen, wie zum Beispiel "Remote-Procedure-Call" oder Nachrichten, kommunizieren.

Die Fähigkeit von Agenten zu migrieren bildet die Grundlage für eine Vielzahl von Anwendungsmöglichkeiten. Beispiele für den Einsatz von mobilen Agenten sind der elektronische Handel (electronic commerce), der Vertrieb von Software, die Unterstützung mobiler Endgeräte und die Dokumentenrecherche. Der Vorteil ist, daß durch das Konzept der mobilen Agenten diese Anwendungsbereiche relativ einfach realisiert werden können.

Betrachtet man sich als Beispiel den Bereich der Informationsrecherche im Internet, so würde bei der Suche nach gewissen Daten zu einem interessierenden Thema ein entfernter Zugriff beim Server erfolgen und alle gefundenen Daten an den Client transportiert. Auf der Client Seite würde dann die letzte Auswahl den Daten (Selektion) durchgeführt.

Durch den Einsatz von mobilen Agenten bei der Informationsrecherche können spezielle "Filter - Agenten" eingesetzt werden, die schon auf der Server Seite, aus der Menge der Informationen die sie erhalten, redundante Daten entfernen und nur die wichtigsten Daten sich raussuchen und an den Client übertragen. Die Menge der transportierten Informationen von der Server- zur Clientseite kann so reduziert werden, indem die Auswahl der Daten auf der Serverseite sich abspielt, bevor die Daten übers Netz an den Client geschickt werden.

Weiterhin muß die Clientseite nicht permanent eingeschaltet sein, während der Agent Daten sammelt. Mobile Agenten benötigen keine dauernde Verbindung zu ihrem Client (Auftraggeber) und arbeiten asynchron. Sie nehmen erst nach Vollendung ihrer Arbeit wieder Kontakt mit ihrem Auftraggeber auf. Die Kommunikationskosten zwischen der Client- und Serverseite kann so reduziert werden.

Am IPVR (Institut für Parallele und Verteilte Höchstleistungsrechner) der Universität Stutt-

gart wurde in der Abteilung Verteilte Systeme, das Agentensystem "**Mole**" ([Baumann et al. 97/98]) konzipiert und implementiert. Mole ist kein Akronym, sondern einfach der Name des Projektmaskottchens als auch der Name des Agentensystems. Mole ist das erste mobile Agentensystem das in der Programmiersprache Java von SUN Microsystems entwickelt wurde. Die Verwendung von Java ist ein wichtiger Schritt, mobile Agenten einem breitem Publikum zugänglich zu machen. Die Verbreitung von Java nimmt ständig zu, und zukünftig werden Java Applets auf allen gängigen Rechnern unabhängig vom verwendeten Betriebssystem, verwendet werden können.

## 1.2 Aufgabenstellung

Eine Vorbedingung zum Einsatz der Agententechnologie im kommerziellen Umfeld ist das Vorhandensein von Mechanismen zur Steigerung der Fehlertoleranz, welche die Ausführung des Agenten auch unter Vorhandensein von Fehlern, sowohl von Netzwerk als auch von ausführenden Rechnern, sicherstellen.

In dieser Diplomarbeit "Fehlertoleranz in Mole" soll das Agentensystem Mole, mit dem mobile Agenten eingesetzt werden können, um solch einen Mechanismus erweitert werden. Grundlage dieser Arbeit ist die vorliegende Implementierung eines fehlertoleranten Protokolls ([Friedel 98]) zur Sicherstellung der Exactly-Once-Ausführung von Agenten. Dieses soll in dieser Arbeit in Mole 3.0 integriert werden.

Die grundlegende Idee des fehlertoleranten Protokolls ist, daß die Ausführung eines Agenten auf einem Knoten (Arbeiter) durch mehrere andere Knoten (Beobachter) überwacht wird. Die vorliegende Implementation des Protokolls erwartet, daß der Arbeiter und die Beobachter (der nächsten Stufe) bei jeder Migration des Agenten explizit angegeben werden.

Die in einer parallelen Arbeit ([Buschle 99]) in Mole integrierte Erweiterung, welche eine sehr flexible Möglichkeit der Agenten-Reiserouten-Spezifikation zur Verfügung stellt, erlaubt eine weitestgehend automatische Bestimmung des Arbeiters und der Beobachter für die nächste Migration.

Für diese automatische Bestimmung ist ein Algorithmus zu entwerfen, dessen Anforderungen an die Reiserouten - Spezifikation (als Vorgabe für deren parallel laufende Implementation) zu spezifizieren und den Algorithmus (StageConstruction) zu implementieren.

Die Implementierung der geforderten Aufgabe erfolgt in den Programmiersprachen Java und C++. Weiterhin ist eine Einarbeitung in das Mobile - Agentensystem Mole als auch Exactly-Once-Protokoll notwendig. Als Java - Entwicklungsumgebung wird JDK 1.1.6 verwendet. Die Arbeit ist auf Workstations vom Typ Sun Sparc unter Solaris 2.6 durchzuführen.

Da das gegebene Protokoll Transaktionen verwendet, wird hierfür der CORBA (Common Object Request Broker Architecture) Transaction Service verwendet. Zu beachten sind bei der Implementierung die Projektrichtlinien der Abteilung. Die Ergebnisse der Arbeit sind in einem Abschlußvortrag zu präsentieren. Das Agentensystem Mole und das Exactly-Once-Protokoll stellen den Rahmen für diese Arbeit. Eine Übersicht über verschiedene Agentensysteme bietet ([Baumann et al. 97]).

## 1.3 Überblick über die Diplomarbeit

Der weitere Aufbau dieser Arbeit gliedert sich wie folgt:

In **Kapitel 2** wird das Agentensystem Mole 3.0, das dieser Arbeit zugrundeliegt, vorgestellt. Einen Überblick über das Exactly-Once-Protokoll wird in **Kapitel 3** gegeben. Zur sinnvollen Integration des Protokolls, wird in **Kapitel 4** näher auf die Änderungen im Agentensystem eingegangen. In **Kapitel 5** wird der Entwurf des zu implementierenden Algorithmus (zur Bildung der nächsten Stufe) dargelegt. Die Integration des Protokolls ins Agentensystem Mole wird in **Kapitel 6** beschrieben. Mit der Zusammenfassung der Arbeit und dem Ausblick auf noch zu bewältigende Aufgaben endet in **Kapitel 7** die eigentliche Ausarbeitung. In **Anhang "A"** werden die Methoden mit den dazugehörigen Parametern und ihrer Funktion aufgeführt. **Anhang "B"** enthält den Projektplan für diese Diplomarbeit. In **Anhang "C"** befindet sich das Literaturverzeichnis.

## Danksagung

Einen besonderen Dank möchte ich an meinem Betreuer Markus Straßer richten, für die gute Zusammenarbeit sowie den produktiven Diskussionen.

## 2 Agentensystem Mole 3.0

Nachfolgend werden einige wichtige Elemente des Agentensystems Mole vorgestellt. Das Agentensystem Mole liegt momentan in der Version 3.0 vor, welches auch die Grundlage für diese Arbeit ist. Es besteht im wesentlichen aus drei Komponenten: den **Agenten**, den **Engines** und den **Locations**.

Auf diese Komponenten wird nun näher eingegangen. Abbildung 2-1 gibt einen Überblick über das Systemmodell von "Mole":

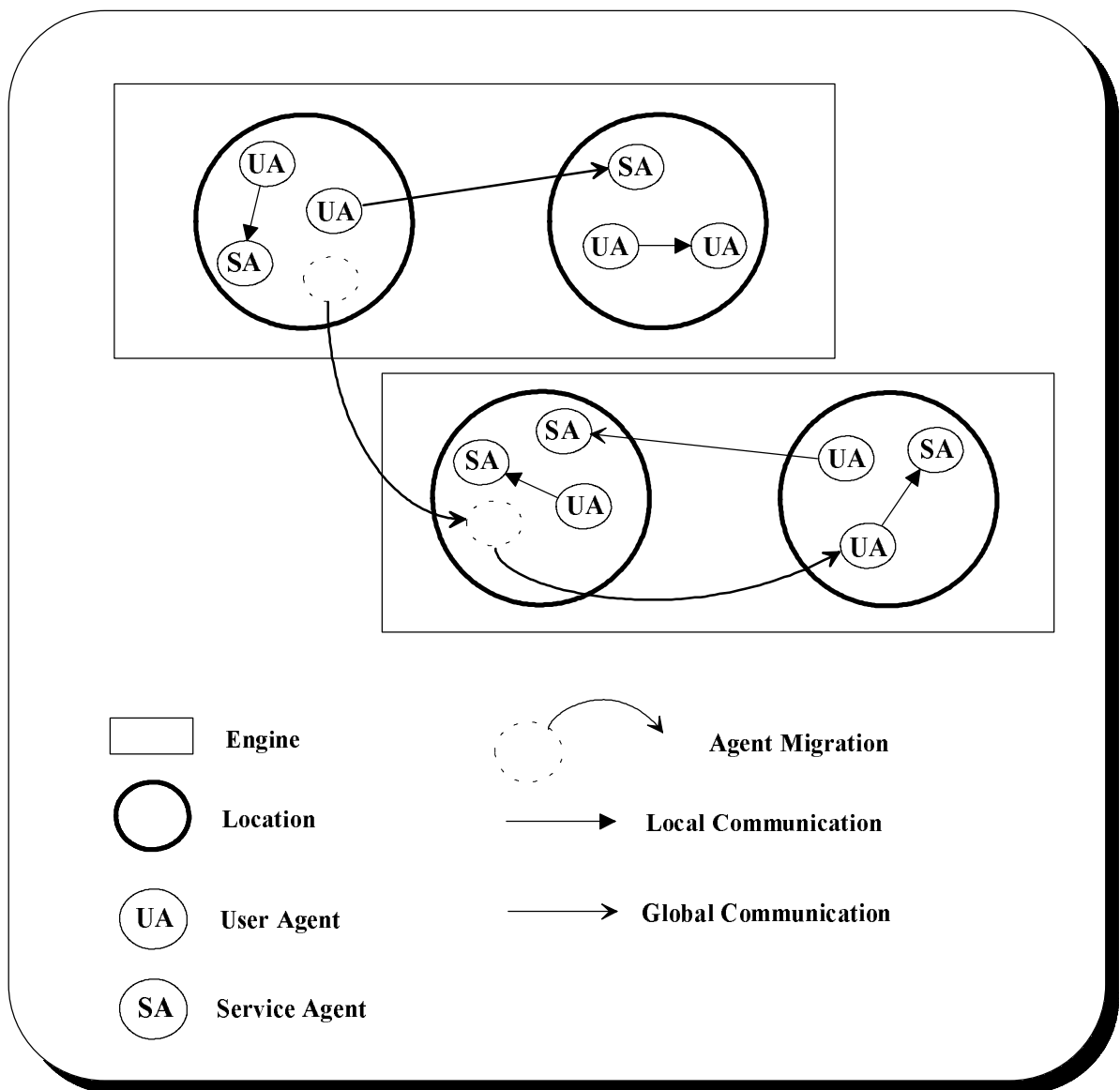


Abbildung 2-1: Systemmodell von "Mole"

## 2.1 Agenten

Agenten bestehen aus Programmcode und Daten. Sie werden über einen global eindeutigen Namen identifiziert. Bei den Agenten in Mole wird im wesentlichen zwischen zwei Typen von Agenten unterschieden: **Benutzer-Agenten** und **System-Agenten**.

### 2.1.1 Benutzer-Agenten

Benutzer-Agenten (User agents) sind aktive Objekte, die die Fähigkeit besitzen, von einer Location zu einer anderen Location zu migrieren und mit anderen Agenten (User agents oder System agents) zu kommunizieren. Deshalb werden sie auch "mobile agents" genannt. Ihnen ist es allerdings nicht erlaubt auf die Ressourcen des Systems zuzugreifen. Sie können aber die Dienste nutzen, die andere Agenten zur Verfügung stellen. Benutzer-Agenten sind diejenigen die von normalen Benutzern des Agentensystems verwendet werden. Solche Agenten können an einer beliebigen Location eingebracht werden und handeln in einem Verteilten System im Namen ihres Benutzers.

### 2.1.2 System-Agenten

System-Agenten (System agents, auch Service agents genannt) bilden gewöhnlich die Schnittstelle zu den Ressourcen (Datenbanken, Festplatten, ...) des Systems. Diese Ressourcen werden von dem System agents als ein Service den User agents zur Verfügung gestellt. Ein User-Agent kann also auf die Ressourcen eines Systems nur zugreifen, indem er mit dem System-Agent kommuniziert. System-Agenten können ausschließlich Services anbieten und befinden sich innerhalb einer Location.

Wenn ein Agent im Agentensystem einen Dienst anbieten möchte, muß er sich bei der Location, auf der er sich befindet, registrieren lassen. Erforderlich ist ein Name für den Service der angeboten werden soll. Dadurch wird anderen Agenten, die einen Dienst in Anspruch nehmen wollen, die Möglichkeit gegeben sich bei der Location mit Hilfe des Namens zu erkundigen, ob auf der Location ein Agent den Dienst anbietet. Sollte der gesuchte Dienst angeboten werden, wird von der Location eine Liste der Agenten zurückgegeben die diesen Dienst zu Verfügung stellen. Ein Agent aus dieser Liste kann dann kontaktiert werden.

Systemdienste und -ressourcen werden somit über System-Agenten von einer Location bereitgestellt.

System-Agenten besitzen allerdings nicht die Fähigkeit zu migrieren wie die User-Agenten. Darum werden sie auch "immobile agents" bezeichnet. Solche System-Agenten können nur von autorisierten Person ins System eingebracht werden.

## 2.2 Kommunikationsmechanismen

In den folgenden Abschnitten sollen die den Agenten (User agents und System agents) zur Verfügung stehenden Kommunikationsmechanismen beschrieben werden. Unter Kommunikation wird der Austausch von Daten verstanden.

### 2.2.1 Messages

Messages (Nachrichten) bieten eine einfache asynchrone Kommunikation an. Dies bedeutet, daß der Thread der eine Message schickt, nach dem absetzen der Nachricht nicht blockiert und somit weitergeführt wird. Eine Message transportiert ein Objekt von einem Agenten zu einem anderen. Das Objekt kann zum Beispiel ein String sein. Die Abbildung 2-2 zeigt die Kommunikationspartner und die Kommunikationsrichtungen zwischen den Agenten.

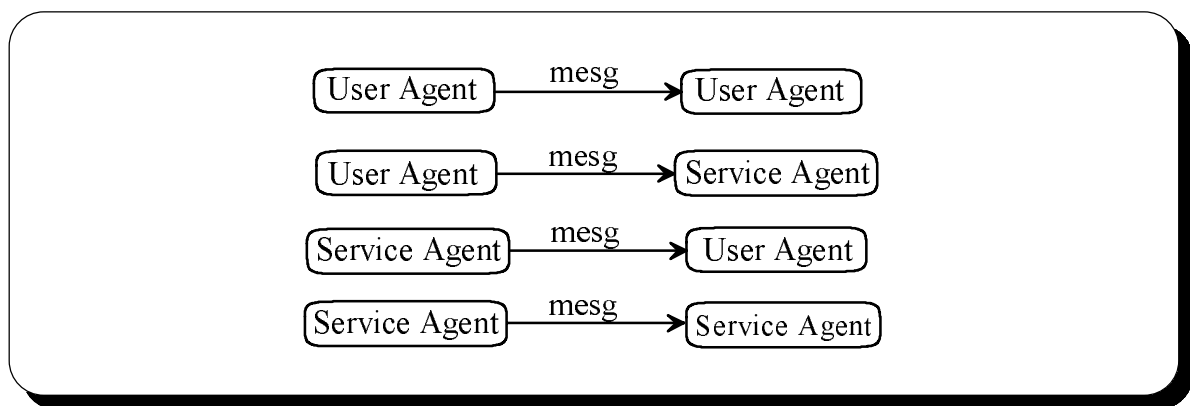


Abbildung 2-2: Kommunikationsmechanismus "Messages"

User agents können sich somit untereinander, als auch ein User-Agent einen Service-Agent und umgekehrt, eine Message ("mesg") schicken. Ebenfalls können Service agents miteinander über "mesg" kommunizieren.

### 2.2.2 Remote Procedure Call

Durch einen RPC (Remote Procedure Call) ist es möglich eine öffentliche Methode eines Agenten aufzurufen. Der Aufruf erfolgt synchron, das bedeutet daß der Thread der einen RPC ausführt solange blockiert wird, bis das Resultat von dem aufgerufenen Agenten zurückgeliefert wird. Bei RPC wird im Gegensatz zu einer Message ein Rückgabewert an den aufrufenden Agenten gesendet, falls vorhanden. In Abbildung 2-3 werden analog die Kommunikationspartner und Kommunikationsrichtungen dargestellt.

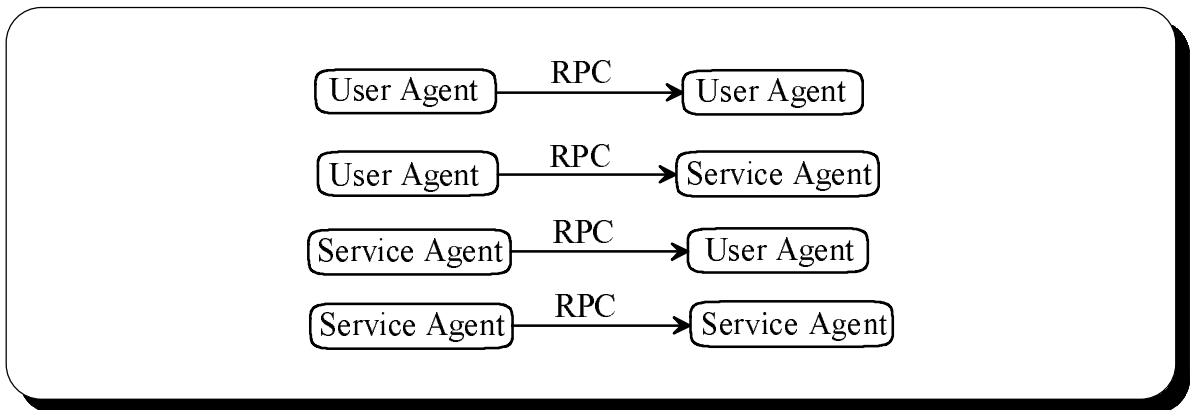


Abbildung 2-3: Kommunikationsmechanismus "RPC"

Ein User-Agent kann wiederum mit einem anderen User-Agenten, als auch ein User-Agent mit einem Service Agent und umgekehrt, über Remote Procedure Call kommunizieren. Es kann ebenfalls ein Service-Agent mit einem Service-Agent über "RPC" kommunizieren.

### 2.2.3 Session

Wenn Agenten über eine längere Zeit miteinander Daten austauschen wollen, können sie eine Session zwischen sich aufbauen. Eine Session ist eine Kommunikationsbeziehung zwischen Agenten. Es wird zuerst eine Session etabliert und dann mittels RPC oder Message Passing kommuniziert. Abbildung 2-4 zeigt die möglichen Kommunikationspartner bei der Session.

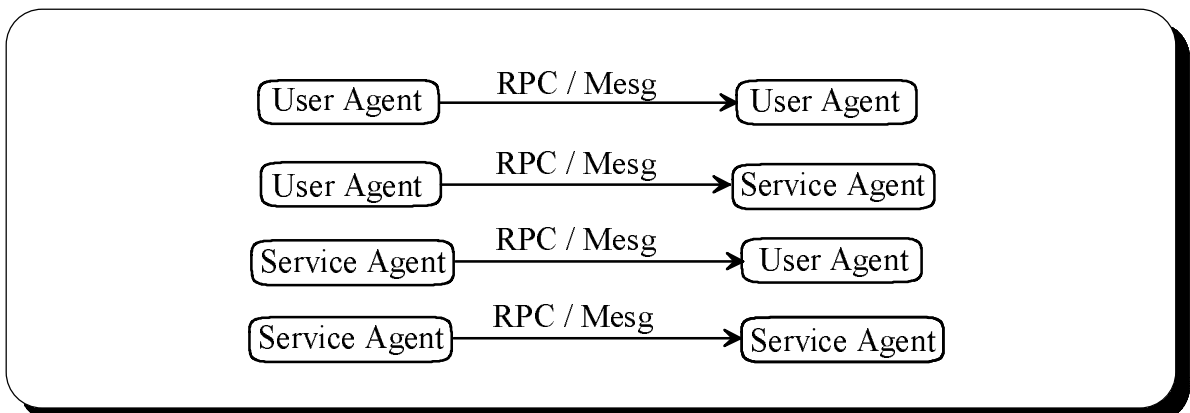


Abbildung 2-4: Kommunikationsmechanismus "Session"

Beim Aufbau einer Session müssen beide Agenten zustimmen. Der Aufbau kann blockierend oder nichtblockierend durchgeführt werden. Nach Beendigung des Datenaustausches wird die Kommunikationsbeziehung zwischen den Agenten aufgelöst.

Die beiden Agenten die bei einer Session teilnehmen, können zu jedem Zeitpunkt die Session explizit oder implizit beenden. Explizit indem einer der beiden Agenten die Methode "terminateSession()" aufruft. Dadurch das ein Agent (der an der Session teilnimmt) zu einer anderen Location migriert, wird die Session implizit beendet. Eine Session kann sowohl, zwischen Agenten innerhalb der selben Location, als auch zwischen Agenten in unterschiedlichen Locations aufgebaut werden. Sie kann verglichen werden mit einem Verbindungsaufbau, Gespräch Verbindungsabbau bei einem Telefongespräch. Mit dem Konzept der Session, können Agenten die sich treffen wollen zum Zwecke der Kooperation, synchronisieren. Bei einer Session kann beschrieben werden, mit welchem Agenten man sich treffen möchte und an welchem Ort.

### **2.2.4 Lokale / Globale Kommunikation**

Messages und Remote Procedure Calls können von Agenten sowohl lokal als auch global benutzt werden. Lokal soll heißen, daß der aufgerufene Agent sich in derselben Location befindet. Global bedeutet, daß der aufgerufene Agent sich auf einer anderen Location aufhält. Die lokale Kommunikation bietet sich an, wenn man Kommunikationskosten reduzieren möchte. Der User-Agent könnte zu einer Location migrieren und sich dort mit einem anderen Agenten treffen und Nachrichten austauschen. Wenn man aber nicht sehr viele Daten auszutauschen hat, kann auch eine Kommunikation, über die Grenzen einer Location stattfinden. Somit können unnötige Migrationen eingespart werden.

Der Programmierer muß sich keine Gedanken darüber machen, ob der Agent der eine Message bekommt oder mit dem mittels RPC kommuniziert werden soll, sich auf der selben Location oder auf einer anderen Location befindet. Die Kommunikation erfolgt jedesmal auf die selbe Weise.

## **2.3 Location**

Eine weitere wichtige Komponente des Agentensystems, ist die Location. Locations sind Orte des Agentensystems, an denen sich Agenten aufhalten, treffen zum Zwecke des Datenaustausches oder um lokale Dienste, die System-Agenten anbieten, in Anspruch nehmen. Die Orte müssen nicht einem physikalischen Rechner entsprechen. Es ist möglich, daß sich mehrere Locations auf einem Rechner befinden.

Eine Location wird identifiziert durch ihren Location - Namen der über DNS (Domain Name System) aufgelöst werden kann. Ein Agent der gerade nicht migriert befindet sich im Augenblick auf einer Location. Da Mole ein offenes System ist, besteht die Möglichkeit daß jeder Benutzer eine neue Location erzeugen kann, und sie der Öffentlichkeit zur Verfügung stellt. Sie ist sowohl für das Starten, Beenden, Migrieren von Agenten, Sicherheit, Auskunftserteilung über Dienste die in der Locations angeboten werden als auch für die zur Verfügungstellung von Methoden für die Kommunikation verantwortlich.

## 2.4 Engine

Die Engine ist die Laufzeitumgebung des Systems. Auf einem Rechner können ein oder mehrere Engines gestartet werden wobei jede Engine ihren eigenen Java - Interpreter benutzt. Eine Engine kann nicht nur eine sondern auch mehrere Locations verwalten. Diese Locations teilen sich den gleichen Java - Interpreter. Die Engine startet die Locations und diese wiederum starten und verwalten die Agenten die zu ihnen gehören. Locations und Agenten werden als Threads gestartet, die vom MCP (Master Control Process) verwaltet werden. Dieser teilt den Agenten Zeitscheiben zu. Da die Engine kein Bestandteil des Agentenmodells von Mole ist, wird sie von den Agenten nicht sichtbar sein.

## 3 Exactly-Once-Protokoll

### 3.1 Einführung

Da im Systemmodell von "Mole" verschiedene Locations auf einem Rechner sein können, wird in diesem Kapitel zur Unterscheidung der Ausdruck "Knoten" verwendet. Mit diesem wird ausgedrückt, daß wenn ein Agent einen Knoten besucht, er sich auf einem Rechner befindet. Auf welcher Location er da ist, ist egal.

In vielen Anwendungen in denen mobile Agenten eingesetzt werden können, wie zum Beispiel im "Electronic Commerce", wird verlangt daß ein Agent genau einmal (Exactly-Once) ausgeführt wird, unabhängig von irgendwelchen Kommunikations- oder Knotenfehlern. Dies bedeutet, daß ein Agent nicht verloren gehen darf, solange seine Ausführung, also die Erfüllung der Aufgabe, die ihm übertragen wurde, nicht abgeschlossen ist.

Ein Agent wird im Normalfall um eine Aufgabe zu erfüllen, mehrere Teilaufgaben durchführen müssen. Dazu kann es eventuell notwendig sein, mehrere Knoten zu besuchen. Diese Knoten können verschiedene Dienste anbieten die von dem Agent zur Bewältigung seiner Aufgabe in Anspruch genommen werden. Die Zuordnung von Teilaufgaben zu Knoten kann mittels einer Reiseroute erreicht werden, die ein Programmierer spezifiziert.

Als Beispiel dient eine bevorstehende Reise von einer Person, die einen Agenten beauftragt eine Flug- und Hotelreservierung vorzunehmen. Der Agent hat als erstes auf Knoten "A" ein Flugticket zu reservieren, dann soll er sich zum Knoten "B" bewegen und dort eine Hotelreservierung vornehmen. Gewährleistet werden muß, daß der Agent auf den Knoten die er besucht um seine Teilaufgaben zu erfüllen genau einmal ausgeführt wird. Er soll ja bei Knoten "A" nur einmal ein Flugticket reservieren. Ohne eine solche Gewährleistung würde niemand einen Agenten mit solch einer Aufgabe beauftragen.

Die Ausführung eines Agenten erfüllt die Exactly-Once-Eigenschaft, wenn die ganze Sequenz von Teilaufgaben abgeschlossen wird und die Menge aller Operationen (im folgenden als **Step** bezeichnet), die für eine Teilaufgabe bei einem Knoten ausgeführt werden, auch nur einmal ausgeführt werden.

Da es in der aktuellen Version von Mole keine Instanz gibt, die die Ausführung eines Agenten überwacht sind folgende Szenarien denkbar:

- a) Der Knoten auf dem der Agent ausgeführt wird, kann zu einem Zeitpunkt aufgrund eines Ausfalls nicht mehr verfügbar sein. Der Agent ist somit solange blockiert, bis der Knoten wieder verfügbar ist.
- b) Es können Netzwerkpartitionierungen auftreten und der Fall eintreten, daß ein Knoten zum Beispiel "B", der als nächstes besucht werden soll von Knoten "A" aus, nicht mehr erreichbar ist weil er sich nun in einer anderen Partition befindet als "A". Der Agent ist wiederum gezwungen, abzuwarten bis der andere Knoten "B" erreichbar ist. Dies führt wieder zu einer

Blockierung des Agenten auch, wenn es vielleicht möglich wäre die Teilaufgabe auf einen anderen Knoten "C", innerhalb der selben Partition wie "A" auszuführen, und somit eine Blockierung zu vermeiden.

Erforderlich ist also ein Protokoll das sowohl mit solchen Szenarien umgehen kann und somit die Wahrscheinlichkeit einer Blockierung reduziert als auch die Exactly-Once-Eigenschaft erfüllt.

### 3.2 Beschreibung des Exactly-Once-Protokolls

Im folgendem Abschnitt wird ein kurzer Überblick über das Exactly-Once-Protokoll gegeben. Für eine ausführliche Beschreibung zu diesem Thema wird auf ([Friedel 98], [RotStr 98], [Maihöfer 97]) verwiesen.

Durch den Einsatz von transaktionalen Message Queues wird die Exactly-Once-Eigenschaft sichergestellt. Auf das Prinzip einer Message Queue wird im folgenden näher eingegangen. Eine Message Queue dient zur Kommunikation zwischen einem Sender und einem Empfänger. Der Sender einer Nachricht übergibt sie nicht direkt dem Empfänger sondern legt sie in eine Message Queue (Warteschlange) ab. Der Empfänger wiederum holt sich die Nachricht aus der Queue. Dies kann allerdings zu einem späteren Zeitpunkt erfolgen, weshalb es sich hier auch um eine asynchrone Kommunikation handelt. Die Message Queue arbeitet nach dem FIFO - Prinzip (First In First Out): Die Nachricht, die als erstes in die Queue abgelegt wird, wird auch als erstes wieder aus der Queue entnommen. Eine Message Queue ist in der Lage, Nachrichten persistent zu speichern, so daß sie auch nach einem Knotenausfall noch auf stabilen Speicher vorhanden sind. Mit Hilfe einer PUT - Anweisung ist es möglich eine Nachricht in die Queue abzulegen. Die Anweisung ist nicht blockierend für den Sender. Er kann fortlaufend Nachrichten in die Queue ablegen auch wenn der Empfänger nicht verfügbar sein sollte. Eine GET - Anweisung wird zum Entfernen einer Nachricht aus der Queue verwendet. Der Empfänger wird nur dann blockiert, wenn bei der Ausführung der GET - Anweisung keine Nachrichten in der Queue vorhanden sind.

Bei der transaktionalen Message Queue werden die beiden Anweisungen innerhalb einer Transaktion ausgeführt. Dabei wird gewährleistet, daß wenn die Transaktion abgeschlossen wird, sowohl die PUT als auch GET Anweisung mit Erfolg ausgeführt wurden. Sollte die Transaktion abgebrochen werden, wegen eines Knoten oder Transaktionsfehlers, so werden alle Änderungen rückgängig gemacht.

Für die Queue ist ein sogenannter Queue Manager verantwortlich der bei einer Akzeptierung einer Nachricht auch dafür sorgt, daß sie einmal ausgeliefert wird, unabhängig von irgendwelchen Kommunikations- oder Knotenfehlern.

Anstatt Nachrichten wird nun im Protokoll ein Agent in die Queue abgelegt, der von Knoten zu Knoten migriert gemäß seiner Reiseroute. Abbildung 3-1 zeigt die Anwendung von transaktionalen Message Queues bei Agenten zur Erfüllung der Exactly-Once-Eigenschaft. Dabei migriert ein Agent von Knoten zu Knoten entlang einer Reiseroute [Knoten 1, Knoten 2, ..., Knoten n], wobei ein Knoten mehrmals besucht werden kann. Beim Start des Agenten wird dieser in die "Queue 1" abgelegt und jeder Knoten mit Ausnahme von "Knoten n", wird

die folgende Sequenz von Anweisungen innerhalb einer Transaktion ausführen:  
Begin, Get(Agent), Execute(Agent), Put(Agent), Commit.

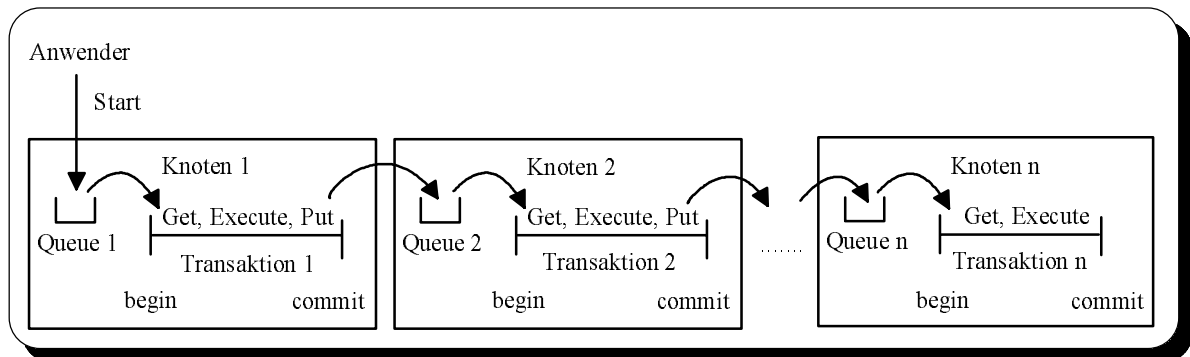


Abbildung 3-1: Transaktionale Message Queues

Mit "Begin" wird die Transaktion gestartet. "Get" holt den Agenten aus der Queue der anschließend von "Execute" ausgeführt wird. Durch "Put" wird der Agent in die Queue des nächsten Knotens abgelegt. Mit "Commit" wird die Transaktion abgeschlossen. Sollte es zu einem Knotenausfall kommen, zum Beispiel bei Knoten 2, so erfolgt ein Transaktionsabbruch und alle Änderungen die den Knoten 2 betreffen, werden rückgängig gemacht. Dies bedeutet, der Agent würde sich wieder in der Queue 2 befinden, wie vor dem Start der Transaktion 2. Die Änderungen betreffen auch die nachfolgende Queue 3 dadurch, daß der Agent aus dieser entfernt wird. Für den Knoten 3 ist der Agent nur dann sichtbar wenn die Transaktion 2 erfolgreich ausgeführt (abgeschlossen) worden wäre. Nachdem der Knoten 2 wieder verfügbar ist, wird die Transaktion 2 erneut ausgeführt. Durch die Transaktion 2 kann der Agent im Fehlerfall zwar mehrmals ausgeführt aber nur einmal mit "Commit" abgeschlossen werden. Somit wird der Agent im Knoten 2 durch ein "Commit" der Transaktion 2, genau einmal ausgeführt.

Erreicht der Agent schließlich den "Knoten n", so wird in "Transaktion n" keine "Put"-Anweisung mehr durchgeführt, nachdem der Agent ausgeführt ist. Dieser wird nach seiner Beendigung aus dem Agentensystem entfernt.

Durch den Einsatz von Message Queues wird allerdings die Blockierung eines Agenten nicht automatisch reduziert. Wenn der Knoten 2 ausfällt bevor die Transaktion 2 abgeschlossen ist, muß der Agent sich solange in der Queue des Knotens 2 aufhalten bis dieser wieder verfügbar ist und die Transaktion 2 den Agenten zur Ausführung aus der Queue holt. Der Agent ist dadurch in der Queue 2 gefangen, auch wenn es andere Knoten gäbe, die verfügbar sind und ihn ausführen könnten. Ein ähnliches Verhalten ergibt sich bei einer Netzwerkpartitionierung, bei der der Agent nicht an die nächste Queue 3 übergeben werden kann, wenn der Knoten 3 durch die Partitionierung sich nicht mehr in der gleichen Partition wie Knoten 2 befindet.

Zur Lösung dieses Problems wurde das Protokoll um sogenannte Observer (Beobachter) ergänzt. Eine Anzahl von Observer überwachen dabei den Knoten der gerade den Step ausführt. Dieser ausführende Knoten wird als Worker (Arbeiter) bezeichnet. Sollte der Worker ausfallen übernimmt einer von diesen Observer die Rolle des Workers und führt den Agenten aus.

Die Observer und der Worker bilden zusammen eine **Stage** (Stufe), in der eine Teilaufgabe bearbeitet wird. Somit wird ein Agent in Stufen ausgeführt.

Abbildung 3-2 soll die Ausführung eines Agenten in 2 Stufen verdeutlichen. Der Anwender schickt einen Agenten auf die Reise. Dabei wird der Agent in alle Queues der Stufe 1 abgelegt. Allen Knoten in einer Stufe wird eine Priorität zugeordnet, so daß eine Totalordnung zwischen ihnen entsteht. Der Knoten mit der höchsten Priorität erhält die Rolle des Workers und die restlichen Knoten die Rolle des Observers. Beim Ausfall des Workers wird der Observer mit der zweithöchsten Priorität die Rolle des Workers übernehmen. Durch die Put-Anweisung des Workers wird der Agent nicht nur zum Worker der nächsten Stufe übertragen, sondern auch zu den Observern damit diese, wenn der Worker ausfällt, eine Kopie vom Agenten besitzen und diesen ausführen können. Mit der Einführung von Observern wird hiermit die Wahrscheinlichkeit einer Blockierung des Agenten reduziert.

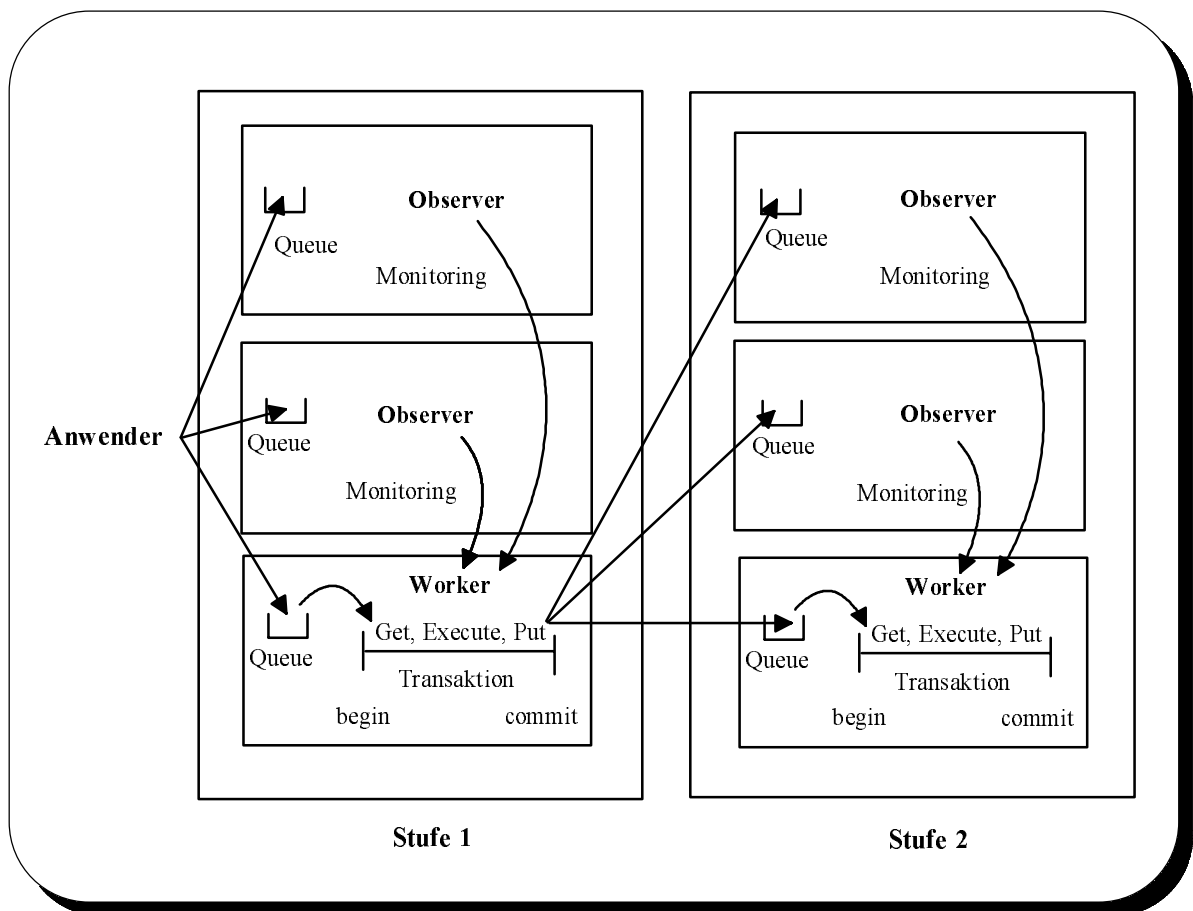


Abbildung 3-2: Agentenausführung in 2 Stufen

Im Exactly-Once-Protokoll sind weitere Protokolle wie Monitoring, Selection und Voting integriert. Das Monitoring Protokoll übernimmt dabei die Überwachung des Workers. Es wird erkannt ob der Worker nicht mehr verfügbar ist. Beim Ausfall des Workers wird durch das Selection Protokoll der Observer bestimmt der als nächstes die Rolle des Workers eingeht.

Dies ist erforderlich da mehrere Observer den Ausfall des Workers erkennen können und versuchen sich selbst als neuen Worker in der Stufe einzuführen. Da ein Observer nicht erkennen kann ob ein Knoten ausgefallen oder in einer anderen Partition (Netzwerkpartitionierung) aktiv ist, kann der Fall eintreten, daß es dann mehrere Worker gibt. Damit aber sichergestellt werden kann, daß nur ein Worker der Stufe ein Commit durchführt und somit die Exactly-Once-Eigenschaft gewahrt bleibt, ist das Voting Protokoll eingeführt worden.

Dabei darf ein Worker der Stufe nur dann ein Commit ausführen, falls dieser eine Mehrheit an Stimmen von den anderen Knoten der Stufe erhält. Bei einer Stufe mit 5 Knoten die in 2 Partitionen (A,B) aufgeteilt wurde mit einmal 3 Knoten in der Partition "A" und 2 Knoten in der Partition "B" müßte ein Worker in der jeweiligen Stufe mindestens 3 Stimmen erhalten um ein Commit durchzuführen zu können. Mit Hilfe des Exactly-Once-Protokolls wird die Exactly-Once-Eigenschaft für Agenten sichergestellt und die Wahrscheinlichkeit für eine Blockierung reduziert. Dies wird durch die Einführung von Observern und transaktionalen Message Queues erreicht.

### 3.3 Anmerkungen

Wenn der Worker einen Agenten ausführt, kann dieser mit einem anderen Agenten (zum Beispiel Service Agent) kommunizieren. Dieser muß vom Prinzip her eine Art Transaktionsressource sein und im 2PC (2 - Phasen - Commit)- Protokoll teilnehmen. Das 2PC wird gestartet wenn ein Worker die Transaktion mit Commit beendet.

In der ersten Phase wird dabei vom Transaction - Manager, an alle Ressourcen die an der Transaktion beteiligt sind, ein "Prepare" gesendet damit diese ihre Änderungen dauerhaft machen. Je nachdem ob die Ressourcen die Änderungen dauerhaft machen können oder nicht wird entweder ein "Yes" oder "No" an dem Initiator des 2PC also dem TM (Transaction - Manager) zurückgegeben. Durch eine "Yes" - Antwort an den TM garantiert eine Ressource, die Änderungen unabhängig von irgendwelchen Knotenfehlern durchführen zu können. Wenn alle Ressourcen mit "Yes" geantwortet haben, wird der Transaction - Manager in der zweiten Phase an alle beteiligten Ressourcen ein "Commit" senden damit diese nun endgültig ihre Änderungen durchführen. Sind die Änderungen durchgeführt, antworten sie dem Transaction Manager mit "Ok". Im Falle das auch nur eine Ressource ein "No" zurückgibt, wird der TM in der zweiten Phase ein "Abort" an die Ressourcen senden, wobei damit die Transaktion abgebrochen wird und die Ressourcen aufgefordert werden ihre Änderungen rückgängig zu machen.

Die Ressourcen antworten wiederum mit einem "Ok", wenn die Änderungen rückgängig gemacht wurden. Eine Transaktion wird somit mit dem 2PC abgeschlossen, wobei garantiert ist, daß alle Ressourcen konsistent zum selben Transaktionsausgang kommen. Die Änderungen die innerhalb einer Transaktion ausgeführt worden sind, sind entweder nach einer Transaktion entweder alle durchgeführt oder wenn die Transaktion abgebrochen wird, rückgängig gemacht.

Die Notwendigkeit, daß ein Service Agent eine Transaktionsressource sein muß, ergibt sich aus dem folgenden Beispiel:

Der Worker der Stufe "i" führt einen Agenten aus und dieser kommuniziert mit einem Service

Agenten, damit dieser eine Überweisung durchführt. Wenn sich nun der Service Agent nicht in der Transaktion befindet, die Überweisung durchgeführt wurde und der Worker ausfällt bevor Commit ausgeführt wird, dann wird die Transaktion abgebrochen und der Agent befindet sich wieder in der Eingangsqueue des Workers, die Überweisung bleibt jedoch bestehen. Die Transaktion ist solange nicht abgeschlossen bis der Agent von der Eingangsqueue der Stufe "i" in die nächste Stufe "i+1" übergeben wird. Ein Observer der den Ausfall des Workers erkennt übernimmt die Rolle des Workers und führt den Agent erneut aus. Somit wird die Überweisung erneut ausgeführt was nicht im Interesse des Kunden ist. Der Vorteil ist nun wenn der Service Agent in der Transaktion teilnimmt, daß nur wenn der Agent von der Eingangsqueue der Stufe "i" in die Queue der nächsten Stufe "i+1" kommt, es garantiert ist das der Vorgang mit der Überweisung, auch nur einmal erfolgreich ausgeführt wird.

Die Teilnahme der Transaktionsressource am 2PC hat zur Folge, daß die Transaktionsressource den Transaktionskontext erhalten muß.

## 4. Mole API

### 4.1 Kommunikation

#### 4.1.1 Partner

##### Kommunikations - Partner in Mole 3.0

Der implementierte Prototyp des Protokolls ([Friedel 98]), zur Sicherstellung der Exactly-Once-Ausführung eines Agenten, soll in das momentan aktuelle Agentensystem Mole 3.0 integriert werden. In der vorliegenden Version des Agentensystem ist es möglich, daß Agenten miteinander kommunizieren können zur Erfüllung einer Aufgabe. Abbildung 4-1 zeigt die möglichen Kommunikationspartner in der Mole Version 3.0:

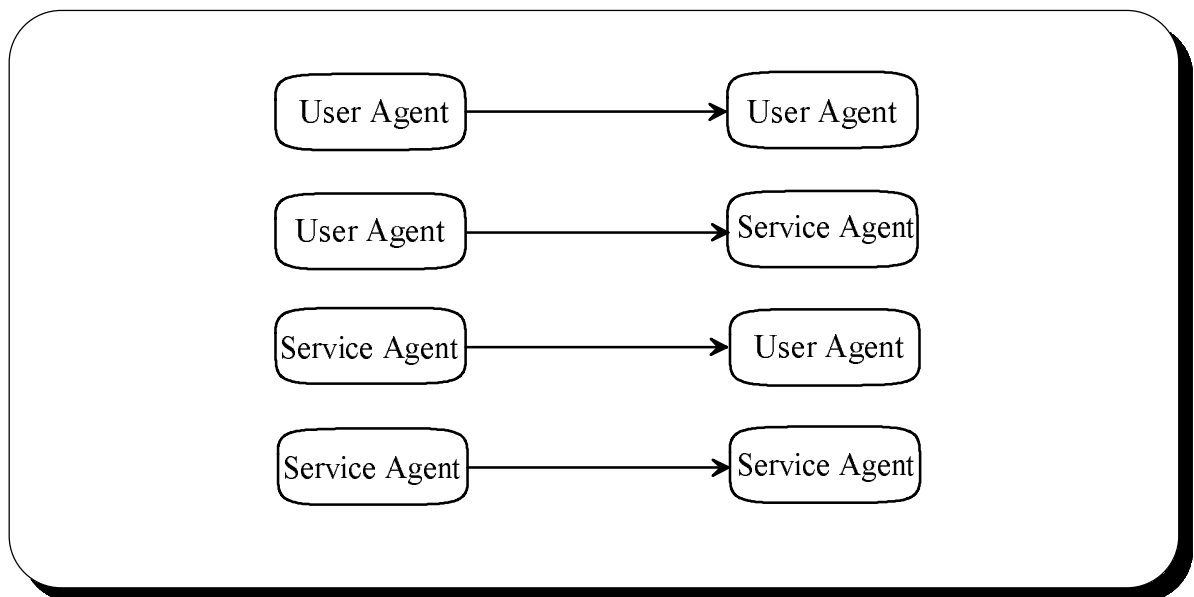


Abbildung 4-1: Kommunikationspartner in Mole 3.0

Dabei können die in Kapitel 2 beschriebenen Kommunikationsmechanismen wie Messages, Remote Procedure Call, Session verwendet werden.

#### 4.1.2 Problem

##### Problembeschreibung

Wenn ein Agent mit anderen Agenten kommuniziert, ist er von anderen abhängig und kann so

prinzipiell blockiert werden. Dies ist nicht wünschenswert, da das Ziel ist, die Reduzierung der Blockierungen zu erreichen.

Deshalb setzt das Protokoll folgende Einschränkung voraus: Ein Agent kann migrieren und sich nur mit Ressourcen (lokale Agenten repräsentieren Ressourcen) unterhalten, nicht aber mit anderen mobilen Agenten. Ein mobiler Agent kommuniziert also ausschließlich nur mit einem Service Agent (lokaler Agent), der einen Dienst zur Verfügung stellt. Ein Service Agent kann wie gewohnt mit einem anderen Service Agent kommunizieren, nicht aber mit einem User-Agenten. Die Abbildung 4-2 verdeutlicht dies nochmals:

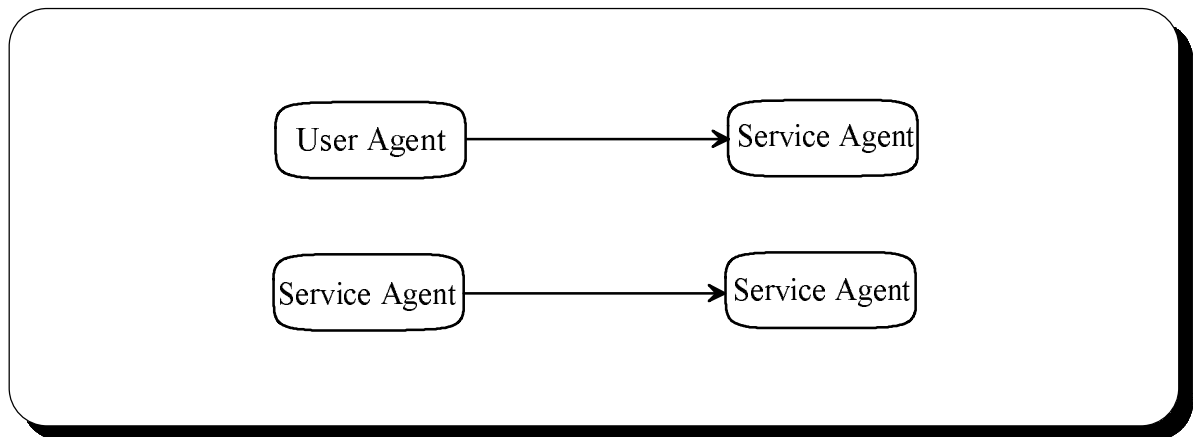


Abbildung 4-2: Kommunikationspartner nach dem Protokoll

Die Folge ist, daß das Protokoll das Ausführungsmodell eines Agenten wesentlich einschränkt, und somit auf der Seite von Mole Anpassungen erforderlich sind. Bezüglich den Kommunikationspartnern sind also Änderungen in der Mole - API (Applications-Programming-Interface) durchzuführen.

### 4.1.3 Einschränkungen

#### Einschränkung der Kommunikations - Mechanismen

Außer den Kommunikationspartnern sind bezüglich den Kommunikationsmechanismen folgende Einschränkung zu machen:

Mobile Agenten dürfen sich mit Service agents nur über den Kommunikationsmechanismus "**Remote Procedure Call**" unterhalten. Der Grund liegt darin, wenn der mobile Agent eine Nachricht an den Service Agent (Ressource) schickt, würde von Paradigma her dies nicht heißen, daß der mobile Agent auf eine Antwort warten muß, was aber durchaus heißen könnte, daß der Worker (der den mobilen Agenten ausführt) eventuell ein **Commit** durchführt, bevor der Service Agent fertig ist, mit dem was in der Transaktion durchzuführen ist. Dann könnte man durchaus Probleme bekommen. Der Service Agent soll zum Beispiel eine Überweisung durchführen, wenn aber der Worker ein Commit durchführt bevor der Service Agent

die Überweisung ausgeführt hat, dann führt dies zu einer Inkonsistenz der Daten. Eventuell kann sogar die Transaktion abgebrochen werden.

Es kann in der transaktionalen Version von Mole eine lokale als auch globale Kommunikation über RPC erfolgen. Allerdings hat die globale Kommunikation über RPC einen Nachteil.

Wenn man eine globale Kommunikation erlaubt, hat man eventuell noch eine weitere Location die an der Transaktion teilnimmt. Dies wäre dann der Fall, wenn die Location (auf der sich der Service Agent befindet und der Remote Procedure Call sich bezieht) einer der Observer der aktuellen Stufe ist (der Worker der aktuellen Stufe führt den Agenten aus von dem der RPC ausgeht) oder sich außerhalb der nächsten Stufe befindet.

Damit ist die Wahrscheinlichkeit, daß eine Location (Knoten) abbricht und damit die Transaktion abbricht auch höher. Je mehr Knoten an der Transaktion teilnehmen, desto größer ist auch die Wahrscheinlichkeit, daß einer abbricht.

Es kann aber auch ein globaler RPC zu einer Location innerhalb der nächsten Stufe vorkommen, dann aber hätte man diesen Nachteil nicht, denn es würde dann keine weitere Location an der Transaktion hinzukommen. Es ist aber eher wahrscheinlich, daß der globale RPC zu einer Location außerhalb der nächsten Stufe erfolgt.

Weiterhin soll eine RPC Kommunikation zwischen Service agents möglich sein. Wenn Agenten miteinander kommunizieren wollen kann es eine Session ([Baumann et al. 97]) geben, bei der dann mit RPC und nicht mit "Message passing" zwischen den Agenten interagiert wird. In der Mole Version 3.0 ist sowohl RPC als auch "Message passing" bei einer Session erlaubt.

### Anforderungen an die Mole API

Es wird eine Spezialversion von Mole 3.0 entworfen, bei der mobile Agenten sich ausschließlich mit Service agents und nicht mit anderen mobilen Agenten unterhalten dürfen. Service agents werden sich mit Service agents unterhalten können, nicht aber mit User agents. In beiden Fällen darf nur über den Kommunikationsmechanismus RPC eine Kommunikation erfolgen. Abbildung 4-3 zeigt dies auf:

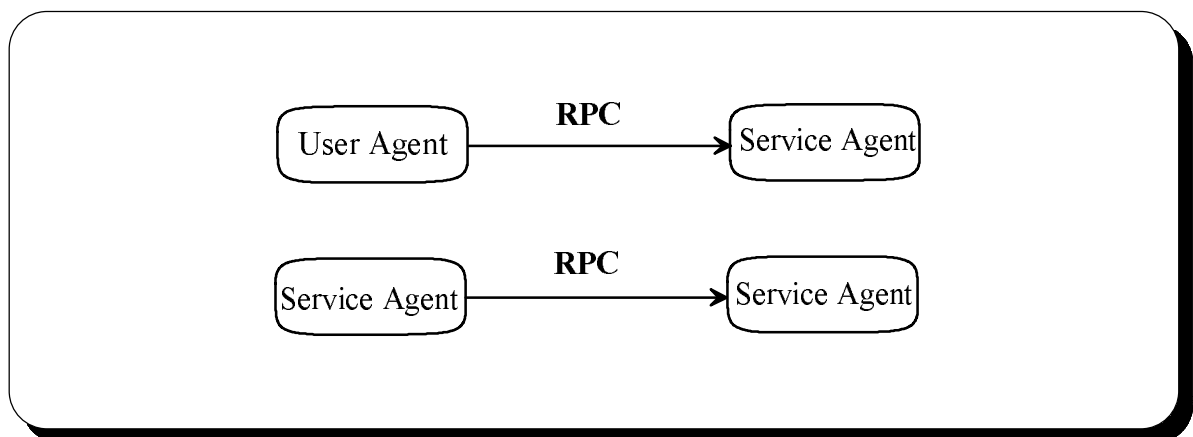


Abbildung 4-3: "RPC" Kommunikation

Wird zur Kommunikation eine Session aufgebaut, so darf auch hier nur mittels RPC kommuniziert werden. Abbildung 4-4 stellt dies dar.

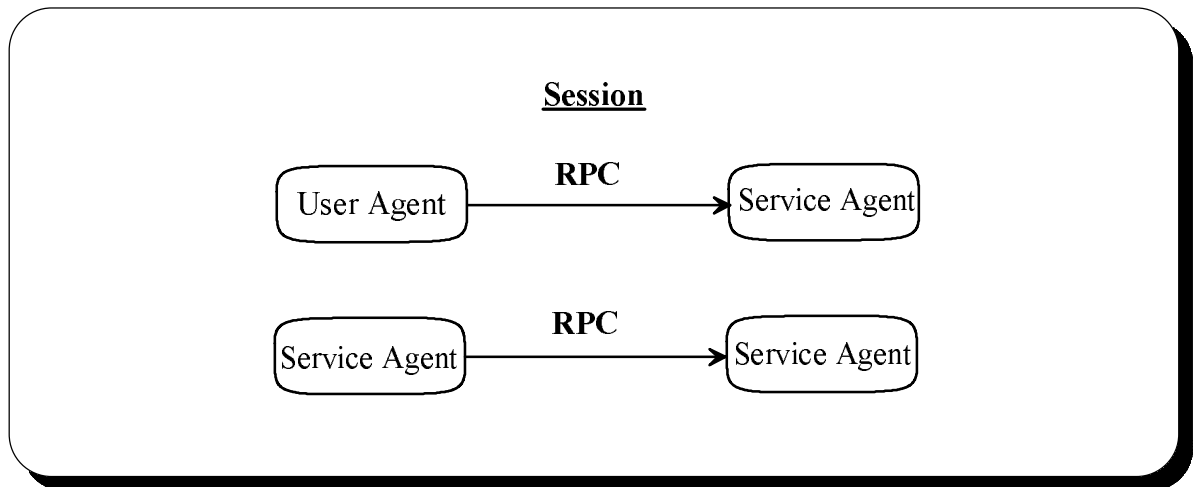


Abbildung 4-4: Session - RPC

Wenn ein RPC ausgeführt wird, muß der Empfänger ein Service Agent sein. Dieser muß wie in dem Kapitel 3 beschrieben, eine Transaktionsressource sein und im 2PC (2 - Phasen - Commit)- Protokoll teilnehmen. Daraus ergibt sich an die Mole - API eine weitere Anforderung; und zwar muß dafür gesorgt werden, daß die Transaktionsressource den Transaktionskontext erhält.

## 4.2 Agentenerzeugung

### Einschränkung der Agenten - Erzeugung

Wenn ein Agent einen anderen Agenten erzeugen will, dann würde dies innerhalb der Transaktion, in der ein Step ausgeführt wird, erfolgen. Ein Agent der von einem anderen Agenten erzeugt wurde ist sofort aktiv, wurde also gestartet.

Im Falle daß der Step erfolgreich ausgeführt wird, ist alles in Ordnung. Wenn aber der Step aus welchen Gründen auch immer nicht erfolgreich ausgeführt wird, hätte man das Problem, das dann der erzeugte Agent aktiv ist, während der Step nicht erfolgreich war und die Transaktion zurückgesetzt wird.

Da nicht sichergestellt ist, das der erzeugte Agent aktiv wird, erst nachdem der Step erfolgreich ausgeführt wurde, ergibt sich für die Mole - API eine weitere Anforderung. Innerhalb einer Transaktion, darf ein Agent keine weiteren Agenten erzeugen. Der Agent darf also selber keine weiteren Agenten erzeugen. Die Methoden mit der man einen neuen Agenten erzeugen kann müssen folglich aus der API entfernt werden.

## 4.3 Migration

### 4.3.1 Bisherige Migration

In der aktuellen Mole Version 3.0 kann ein mobiler Agent um eine ihm übertragene Aufgabe zu erfüllen zu einer Location migrieren auf der ein Service Agent einen Dienst zur Verfügung stellt und diesen Service in Anspruch nehmen. Durch den Aufruf der `migrateTo()` - Methode (unter Angabe der Location), kann der Agent zu einer Location migrieren und dort seine Aufgabe (Step) ausführen Dies wird durch den Programmierer des Agenten angegeben.

### 4.3.2 Anwendung der Reiseroute

Die Grundidee beim Protokoll zur Bildung einer Stufe (ohne Informationen von der Reiseroute) ist, daß man in einer Stufe mehrere Knoten hat, die man beliebig miteinander austauschen kann. Zum Beispiel wenn ein Auto reserviert werden soll, dann kann eine Autoreservierung bei der Hauptniederlassung in Stuttgart oder bei der Hauptniederlassung in Frankfurt oder in Berlin vorgenommen werden. Wo die Reservierung vorgenommen wird ist egal. Die Hauptsache ist, daß ein Auto reserviert wird. In einer Stufe soll ein Agent genau eine Aufgabe (Step) erledigen. In diesem Fall hätte der Agent verschiedene Möglichkeiten um seine Aufgabe zu erfüllen.

Die Grundidee beim Protokoll zur Bildung einer Stufe (mit Informationen von der Reiseroute) ist, daß wenn die Reiseroute die Information liefern könnte, daß der Agent zum momentanen Zeitpunkt entweder auf der "Location 1" den "Step A" oder auf der "Location 2" den "Step B" oder auf der "Location 3" den "Step C" ausführen kann, dann könnte man eine Stufe aus diesen Locations zusammensetzen. Sollte "Location 1" ausfallen, könnte als Beispiel auf "Location 2" der "Step B" alternativ ausgeführt werden.

Ohne die Informationen von der Reiseroute kann also in einer Stufe nur ein Step ausgeführt werden, wobei im Gegensatz dazu also mit Informationen von der Reiseroute in einer Stufe auch nur ein Step ausgeführt wird, es aber möglich ist, daß beim Ausfall einer Location eine andere Location innerhalb der selben Stufe, einen **alternativen Step** ausführen kann. Durch die in ([Buschle 99]) realisierte Reiseroute muß keine exakte Reihenfolge festgelegt werden. Sie legt nur irgendwelche Beziehungen fest, zum Beispiel bevor der Agent die Location "Y" besucht muß er die Location "X" besucht haben. Deshalb ist es dann möglich die Reiseroute so zu spezifizieren, daß zu einem beliebigen Zeitpunkt nicht nur eine Möglichkeit besteht wo der Agent als nächstes hingehen kann, sondern beliebig viele.

Wenn ein Worker einen Step ausführt und dieser aufgrund eines Ausfalls nicht mehr verfügbar ist, so kann die Wahrscheinlichkeit einer Blockierung dadurch reduziert werden, daß mit Hilfe der Reiseroute ein alternativer Step auf einer anderen Location innerhalb der selben Stufe vom Protokoll ausgeführt wird. Wenn beim Protokoll von ([Friedel 98]) ein Worker der einen

"Step A" ausführen soll ausfällt, so wird ein Observer die Rolle des Workers übernehmen und diesen "Step A" ausführen. Mit Hilfe der Reiseroute, die alternative Steps zur Verfügung stellt (wenn vorhanden), ist es nun möglich, daß beim Ausfall des Workers, der "Step A" ausführen soll, nun ein Observer zum Worker wird und einen Alternativen "Step B" in der selben Stufe ausführt.

Es ist ersichtlich, daß jetzt die Reiseroute festlegt zu welcher Location ein Agent sich bewegen kann und welcher Step ausgeführt werden soll. Durch die Verwendung des Exactly-Once-Protokolls und einer Reiseroute wird ein Agent in einer Stufe von den alternativen Steps die die Reiseroute liefert genau einen Step ausführen. Ist der Step in einer Stufe abgearbeitet wird der Agent zu den Queues der Locations der nächsten Stufe übergeben. Wenn sich der Agent nun von der Stufe "i" zur "i+1" bewegt entspricht dies einer Migration des Agenten, weshalb es auch keine migrateTo() - Methode in dieser Spezialversion von Mole mehr geben wird. Durch das Protokoll wird veranlaßt die nächste Stufe zu bilden (wofür in dieser Arbeit ein Algorithmus geschrieben wird der im Kapitel 5 beschrieben wird), bei der der Agent den nächsten Step ausführen soll.

Eine Reiseroute besteht aus Entries. Dabei werden B\_Entries und I\_Entries unterschieden ([Straßer 98]). Im folgenden werden diese Begriffe erläutert die bei der Beschreibung der Stage - Construction (Abschnitt 5.6) verwendet werden.

#### **B\_Entry:**

Ein B\_Entry ist ein Eintrag der aus (precondition, location, method) besteht. Location ist dabei der Ort bei der der Step also die "method" ausgeführt wird. Precondition ist eine Bedingung, die erfüllt (also true) sein muß, bevor die "method" ausgeführt wird.

#### **I\_Entry:**

Ein I\_Entry wiederum ist ein Eintrag, das aus mehreren B\_Entries oder I\_Entries bestehen kann. Die Entries werden ausgeführt wenn die Precondition des I\_Entry erfüllt ist. Zwischen den Entries kann es Prioritätsbeziehungen geben.

### **4.4 Entwurf der Mole API**

Die vorliegende API des Agentensystem Mole 3.0 wird dahingehend verändert, daß einige Klassen bezüglich ihres Verhaltens geändert werden müssen, um die oben aufgeführten Anforderungen zu erfüllen. Es werden also Teile der Mole API geändert und das Resultat wird eine eingeschränkte Version von Mole sein. Im folgenden wird ausschließlich nur auf die durchgeführten Änderungen in der Mole API Bezug genommen. Zu diesem Zeitpunkt wird das Exactly-Once-Protokoll noch nicht in Mole eingebaut. Es werden diejenigen Änderungen in der Mole API vorgenommen, die zur sinnvollen Integration des Exactly-Once-Protokolls notwendig sind und sowohl die Kommunikationspartner, die Kommunikationsmethoden, die

Agentenerzeugung und die Migration betreffen. Abbildung 4-5 und Abbildung 4-6 geben einen Überblick über die Teile der Mole - API die verändert werden, dabei ist folgende Notation zu beachten:

- V = Veränderung der Klasse oder der Methode.
- VI = Veränderung einer Schnittstelle.
- L = Löschung der Klasse, der Methode oder der Variablen.
- LI = Löschung einer Schnittstelle
- U = Klasse bleibt unverändert.
- N = Erzeugung der Klasse, der Methode oder der Variablen.

Verwendet werden UML- (Unified Modeling Language) Diagramme ([FowKen 98]). Als Sichtbarkeitsmodifizierer werden in der UML: public (+), protected (#), private (-) und static (\$) verwendet. Eine Ausführliche Beschreibung der Klassen, Methoden, Variablen vom Agentensystem Mole 3.0, sowie die Klassen und Methoden die zusätzlich noch hinzugefügt wurden sind im Anhang "A.1" aufgeführt.

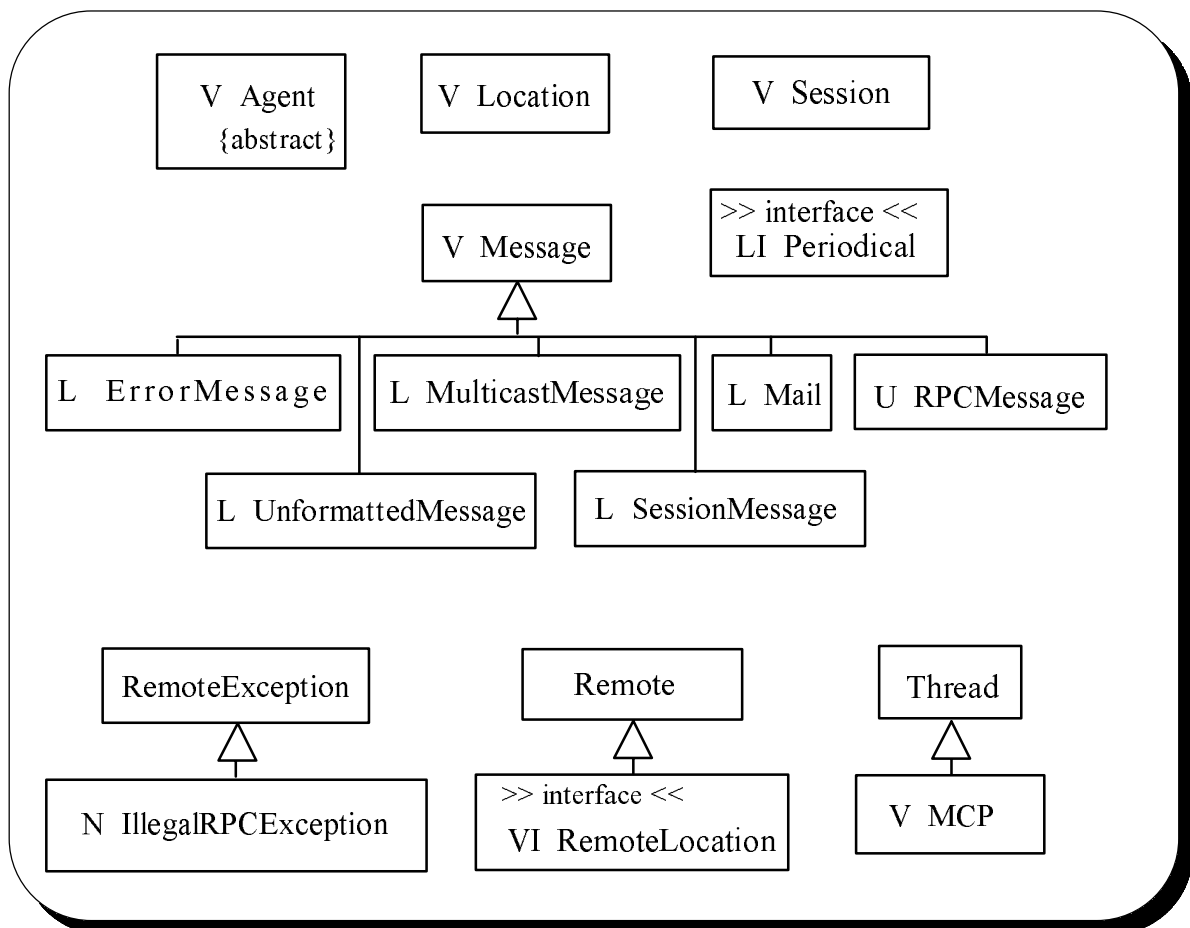


Abbildung 4-5: Eingeschränkte Mole - API (Teil 1)

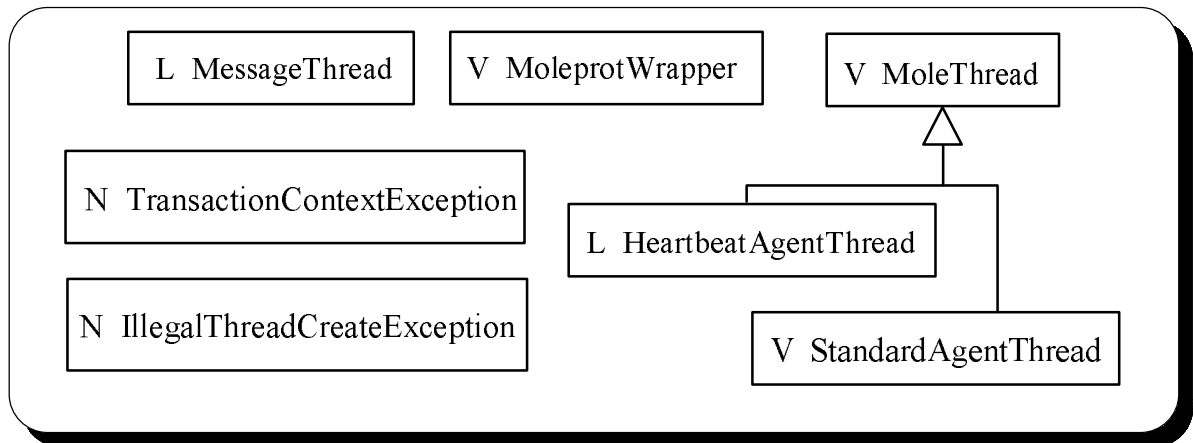


Abbildung 4-6: Eingeschränkte Mole - API (Teil 2)

In den folgenden UML - Diagrammen wird auf die Veränderungen Bezug genommen die einer Erklärung bedürfen.

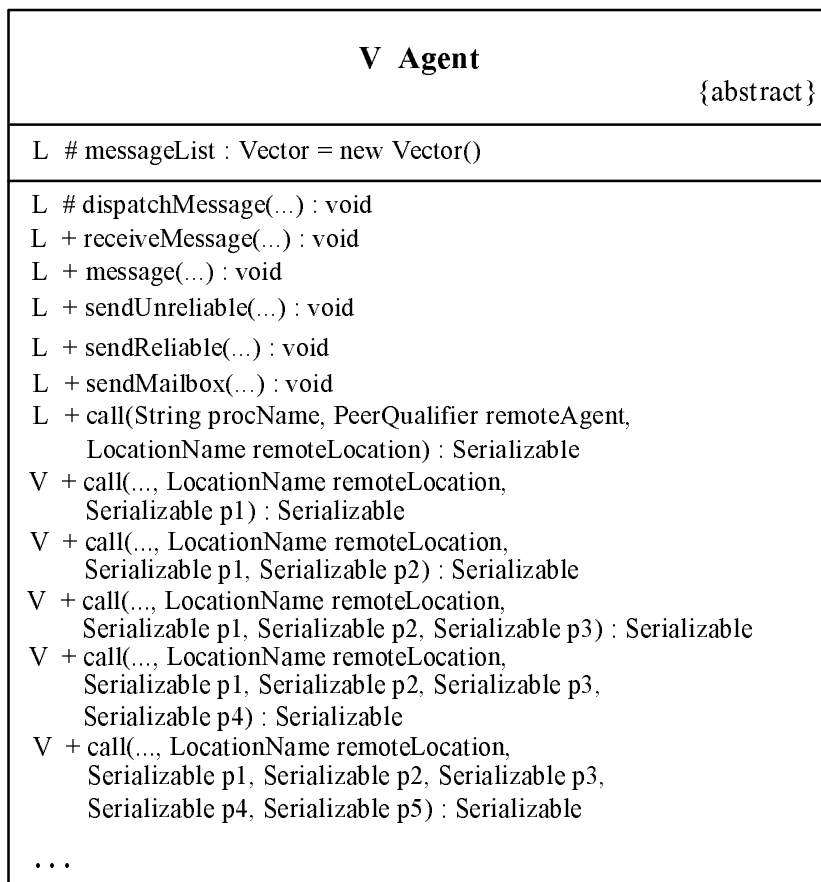


Abbildung 4-7a: Agent - Klasse

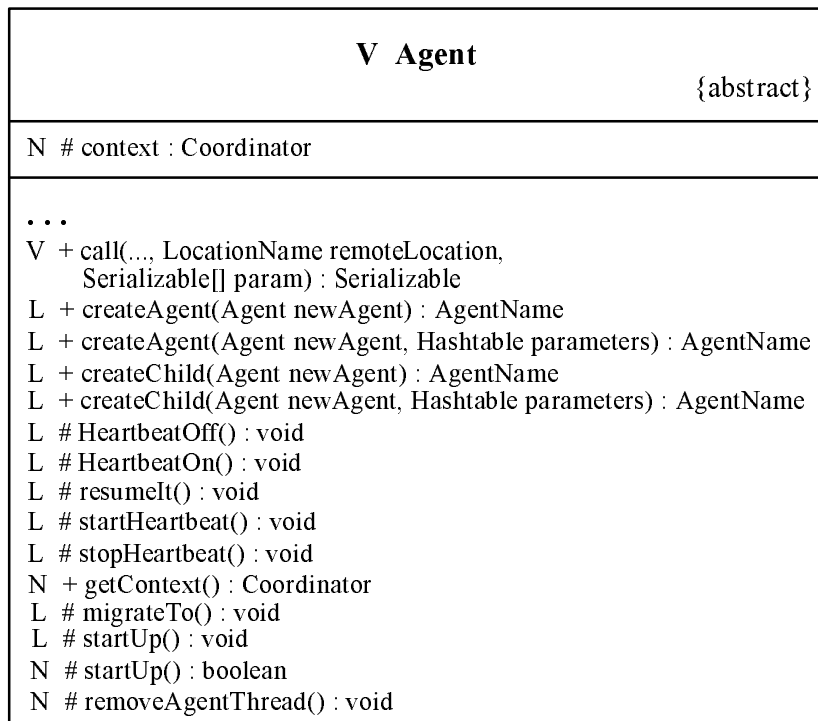


Abbildung 4-7b: Agent - Klasse

### "call(String procName, PeerQualifier remoteAgent, LocationName remoteLocatation)"

Der Transaktionskontext muß dem Service Agent übergeben werden. Dies wird indirekt über den Kommunikationsmechanismus RPC durchgeführt. Dabei wird festgelegt das der erste Parameter der aufgerufenen Methode, den Transaktionskontext (Coordinator) enthalten muß. Wenn also ein mobiler Agent eine call - Methode aufruft, muß als erster Parameter der aufgerufenen Methode der Transaktionskontext (Coordinator) verwendet werden. Zu diesem Zweck wird in der Agent - Klasse die Methode "getContext()" zur Verfügung gestellt die den Coordinator liefert. Alle Methoden die vom Service Agent aufgerufen werden können, werden also den Transaktionskontext erhalten. Der Programmierer von einem Service Agent muß dafür sorgen, daß bei Methoden die aufgerufen werden können, der erste Parameter vom Typ Coordinator ist.

Es ist ersichtlich das die call - Methode, die keinen Parameter hat für die aufgerufene Methode, gelöscht werden muß. Dies gilt auch für die restlichen Klassendiagramme in der eine call - Methode vorkommt die für die aufgerufene Methode keine Parameter enthält. Im nachfolgenden Beispiel wird die Verwendung des Transaktionskontext bei der Benutzung der RPC- Methode (call - Methode) verdeutlicht.

#### Beispiel:

Es soll eine Methode "square" (Coordinator c, Integer i, Float f) aufgerufen werden die eine Zahl zurückliefert. Im Agenten der die call - Methode verwendet wir folgendes stehen:

```

Serializable res = null;
Serializable[] sparam = new Serializable[3];
try
{
    sparam[0] = getContext();
    sparam[1] = new Integer(1);
    sparam[2] = new Float(1.0);
    res = call ("square", remoteAgent, remoteLocation, sparam);
}
catch (Exception e)
{
    Engine.error("Caller: Caught Exception in Caller");
}

```

**"createChild(...), createAgent(...), HeartbeatOff(), HearbeatOn(), startHeartbeat(), stopHeartbeat()"**

Diese Methoden wurden aufgrund Abschnitt 4.2 gelöscht.

**"removeAgentThread()"**

Mit "removeAgentThread()" werden alle Threads des Agenten außer dem aktiven Thread der "removeAgentThread()" ausführt gestoppt.

<b>V Location</b>
L - messages : int = 1000 L - mails : Vector = new Vector(messages) L - messagesrep : Hashtable = new Hashtable()
L message(...) : void L sendUnreliable(...) : void L sendReliable(...) : void L sendMailbox(...) : void L + deliverMessage(...) : void L + syncmessage(...) : void L + numberOfMessages(...) : int L + deliverMessagesFor(...) : void L + setMessagesRepresentative(...) : void L - automessage(...) : void L - storeMessage(...) : void ...

Abbildung 4-8a: Location - Klasse

<b>V Location</b>
<pre> ... V + dispatchRPC(...) : Serializable L + call(AgentName caller, String procName,         PeerQualifier remoteAgent, LocationName remoteLocation) : Serializable V + call(..., LocationName remoteLocation,         Serializable p1) : Serializable V + call(..., LocationName remoteLocation,         Serializable p1, Serializable p2) : Serializable V + call(..., LocationName remoteLocation,         Serializable p1, Serializable p2, Serializable p3) : Serializable V + call(..., LocationName remoteLocation,         Serializable p1, Serializable p2, Serializable p3,         Serializable p4) : Serializable V + call(..., LocationName remoteLocation,         Serializable p1, Serializable p2, Serializable p3,         Serializable p4, Serializable p5) : Serializable V + call(..., LocationName remoteLocation,         Serializable[] param) : Serializable L - lowlevelsend(...) : void V - _arrive(...) : boolean L + setRPCRepresentative(...) : void V + OrphanCall(...) : Serializable L   goTo(...) : boolean L + handleMigration(...) : boolean N + executeAgent(Agent agent) : boolean N   deactivateAgent(Agent traveller) : void V + getMoleview() : MoleviewAgent V # startAgents() : void </pre>

Abbildung 4-8b: Location - Klasse

**"dispatchRPC(...)":**

In dieser Methode wird zusätzlich eine Prüfung vorgenommen, ob der Empfänger des RPC ein User - Agent ist. Trifft dies zu wird eine Exception geworfen. Wenn keine Exception geworfen wird ist somit gewährleistet, daß der Empfänger ein Service Agent ist und ein Agent nur mit diesem kommuniziert.

**"setRPCRepresentative(...)":**

Ein Service Agent migriert nicht, deshalb ist ein Repräsentant von diesem nicht relevant. Für mobile Agenten braucht man auch natürlich keine Repräsentanten weil man mit diesen auch nicht kommunizieren darf.

**"executeAgent(...)"**

Durch "executeAgent(...)" wird der Agent in der Worker Klasse ausgeführt.

**"deactivateAgent(...)"**

Damit die Threads des Agenten alle außer dem aktiven Thread gestoppt werden, wird deactivateAgent(...) verwendet.

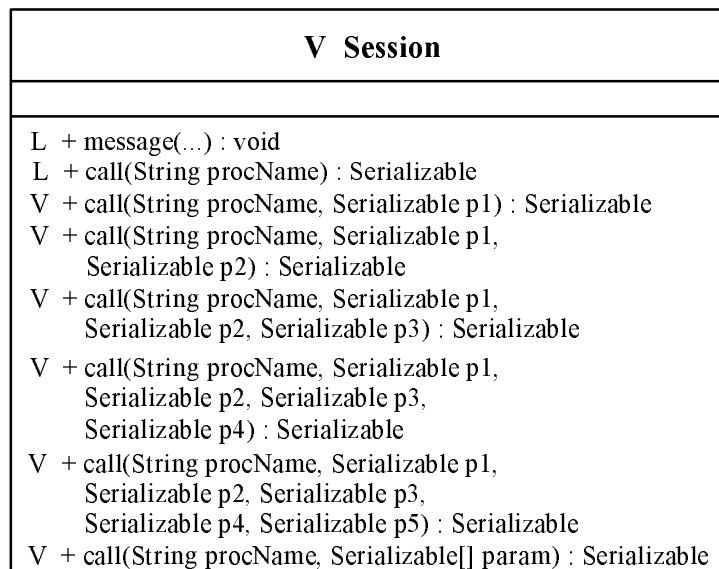


Abbildung 4-9: Session - Klasse

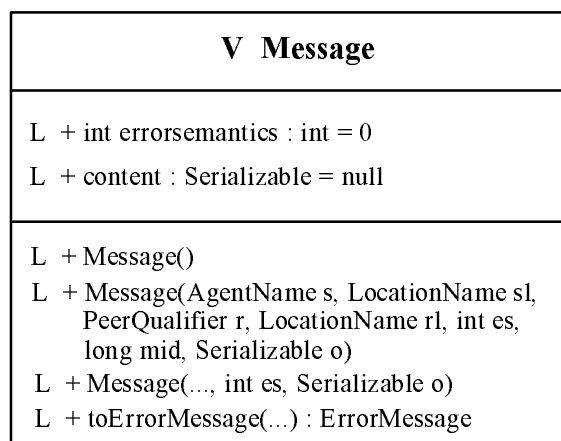


Abbildung 4-10: Message - Klasse

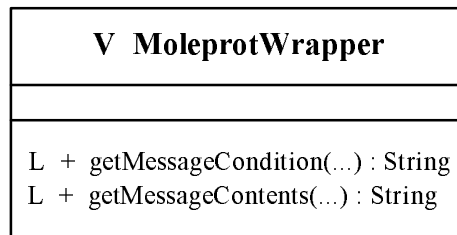


Abbildung 4-11: MoleprotWrapper - Klasse

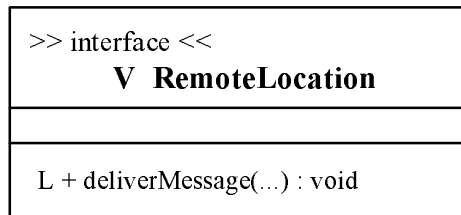


Abbildung 4-12: RemoteLocation - Schnittstelle

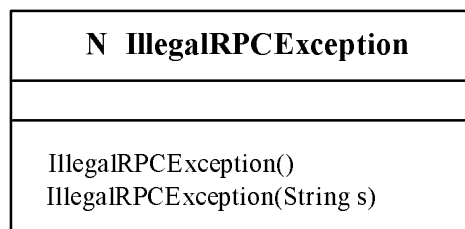


Abbildung 4-13: IllegalRPCException - Klasse

Die Mole API wird weiterhin um die Klasse `IllegalRPCException` erweitert, um eine entsprechende Exception zu werfen, wenn versucht wird mit einem User - Agenten zu kommunizieren.

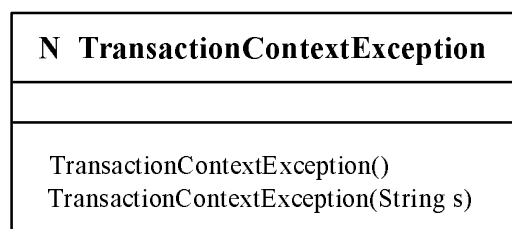


Abbildung 4-14: TransactionContextException - Klasse

Die Klasse `TransactionContextException` wird in die Mole API hinzugefügt, um eine Exception zu werfen, wenn bei der Verwendung einer `call` - Methode, der erste Parameter der aufgerufenen Methode nicht der Transaktionskontext(`Coordinator`) ist.

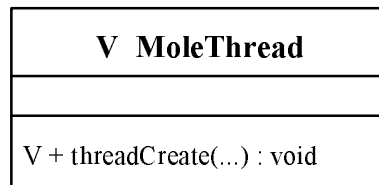


Abbildung 4-15: MoleThread - Klasse

Wenn der mobile Agent in der Step Methode die Methode **threadCreate()** benutzen würde, wird ein Thread gestartet. Dabei kann es passieren, das der Agent in der Transaktion abgebrochen wird, während der Thread noch weiter laufen würde (siehe Abschnitt 4.2).

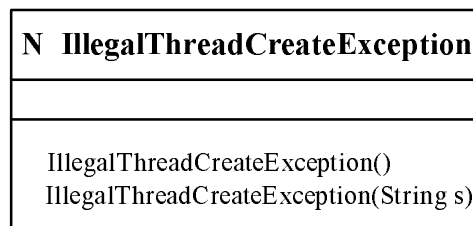


Abbildung 4-16: IllegalThreadCreateException - Klasse

Die Klasse `IllegalThreadCreateException` wird in die Mole API hinzugefügt, um eine Exception zu werfen, wenn ein mobiler Agent versucht "threadCreate" (Abbildung 4-15) zu verwenden.

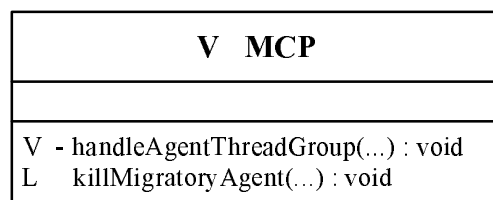


Abbildung 4-17: MCP - Klasse

### "killMigratoryAgent(...)"

Diese Methode wird gelöscht, da sie alle Threads einschließlich den aktiven Thread stoppt, dies jedoch nicht benötigt wird.

<b>Agent : V StandardAgentThread</b>
V + StandardAgentThread() V # runSubclass()

Abbildung 4-18: StandardAgentThread - Klasse

Damit sichergestellt werden kann, daß ein Step des Agenten, der innerhalb eines Threads ausgeführt wird, dieser Thread zuerst ausgeführt wird und dann das Programm des Workers die Bearbeitung fortsetzt, wird hierzu ein Mutex - Semaphor in dieser Klasse verwendet. Die Klasse StandardAgentThread befindet sich in der Agent - Klasse.

## 5. Stage Construction

### 5.1 Vorbemerkungen

Bei der Migration eines mobilen Agenten in der Mole Version 3.0 wird die nächste zu besuchende Location vom Programmierer explizit mit der Methode "migrateTo()" angegeben. Weiterhin ist in der vorliegenden Implementierung des Exactly-Once-Protokolls ([Friedel 98]) vorgesehen, daß die Knoten, die eine Stufe bilden ebenfalls vom Programmierer bestimmt werden. Der Worker und die Observer für die nächste Stufe werden also explizit erwartet.

Durch das Konzept einer Reiseroute ([Buschle 99]), kann der Reiseplan eines Agenten flexibel spezifiziert werden. Die Reiseroute legt fest, zu welcher Location ein Agent migrieren kann. Vom Prinzip her stehen in einer Reiseroute die Locations die der Agent besuchen soll und welche Aufgabe er da zu erfüllen hat.

Da das Exactly-Once-Protokoll in Mole integriert werden soll und es dort keine Knoten sondern Locations gibt, wird im folgenden von Locations die Rede sein.

### 5.2 Interaktion zwischen Protokoll und Reiseroute

Wenn sich ein Agent von einer Stufe "i" zu einer anderen Stufe "i+1" bewegt (dies entspricht in etwa der Migration von einer Location zur anderen), werden mittels des in dieser Arbeit beschriebenen Algorithmus (Beschreibung siehe Abschnitt 5.6) die Locations der nächsten Stufe bestimmt.

Dazu läßt man sich von der Reiseroute eine Liste von Einträgen geben. Diese Einträge bestehen jeweils aus einer Location und einer Methode. Bei der Methode handelt es sich um den Step (Schritt) der in der Location ausgeführt werden soll. Die Steps die man so erhält, können alternativ ausgeführt werden. Diese Einträge werden von dem Algorithmus dazu verwendet um den Arbeiter und die Beobachter der nächsten Stufe "i+1" zu bestimmen, also die nächste Stufe zu bilden.

Sobald die Stufe gebildet ist, wird mit der Put - Operation der Transaktion, der Agent zu den Queues dieser Locations in der nächsten Stufe geschickt. Zu diesen Locations wird also eine Kopie des Agenten geschickt. Abbildung 5-1 zeigt den Zusammenhang zwischen dem Protokoll und der Reiseroute.

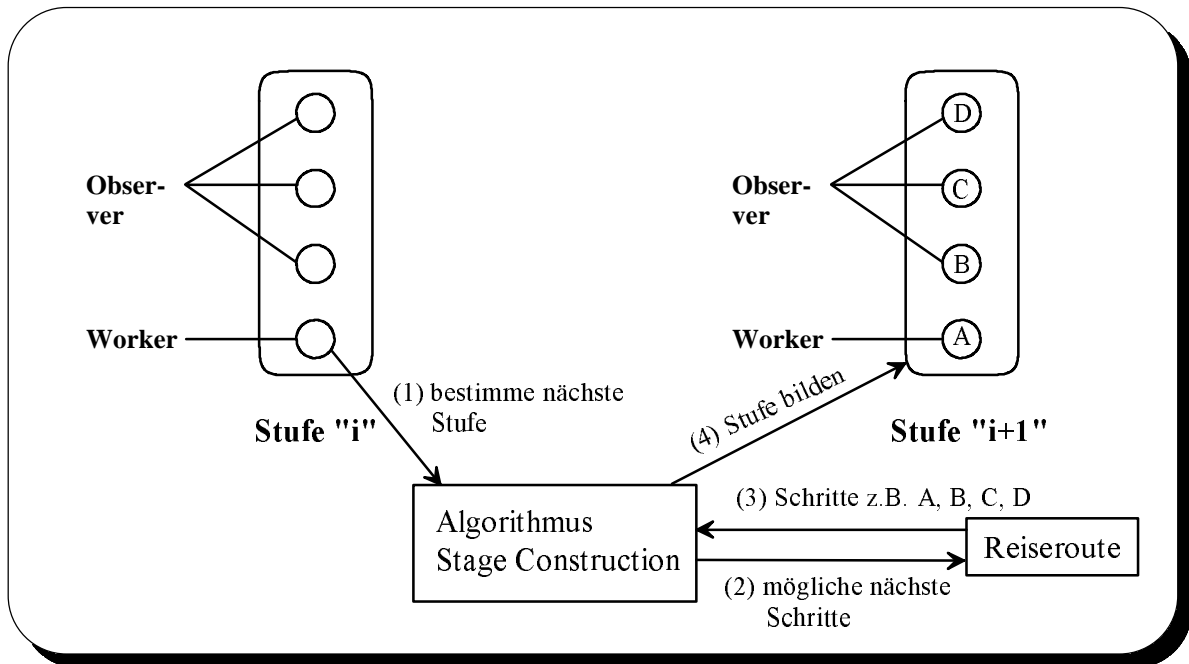


Abbildung 5-1: Stage Construction und Reiseroute

### 5.3 Anzahl der Stage - Locations

Ein wesentlicher Einflußfaktor bei der Zuverlässigkeit (Reliability) für eine Stufe, daß der Agent ausgeführt wird, ist die Anzahl der Locations die innerhalb einer Stufe verwendet werden. Durch das Exactly-Once-Protokoll wird eine Steigerung der Zuverlässigkeit erreicht, im Gegensatz zum einfachen Protokoll bei dem quasi eine Stufe aus einer Location besteht.

Bei einer Untersuchung ([Straßer 98]) der Blockierungswahrscheinlichkeit eines Agenten innerhalb einer Stufe, wurde die Erkenntnis gewonnen, daß eine Stufe aus einer ungeraden Anzahl von Knoten und mindestens 3 Knoten bestehen sollte. Dadurch wird die Blockierungswahrscheinlichkeit eines Agenten erheblich reduziert.

Um diese Erkenntnisse zu berücksichtigen, wird in dem Stage - Construction Algorithmus zur Bildung der nächsten Stufe, eine Anzahl von 3 Locations als Defaultwert für eine Stufe festgelegt.

### 5.4 Typen von Stage - Locations

In einer Stufe können zwei Typen von Locations unterschieden werden. Zum einem sind es die sogenannten **regular locations** (reguläre Orte). Diese zeichnen sich dadurch aus, daß Agent gemäß Reiseroutenspezifikation auf ihnen zum aktuellen Zeitpunkt einen Step ausführen kann (zum Beispiel ein Flugticket reservieren). Auf der anderen Seite es die sogenannte **exception**

**handling locations** (Ausnahmebehandlungs-Orte) die dem Agenten nur eine Laufzeitumgebung zur Verfügung stellen. Ein Agent wird nur dann einen Step bei solch einem Ort ausführen wenn alle regular locations der Stufe nicht verfügbar sind, zum Beispiel aufgrund eines Ausfalls. In diesem Fall wird auf einer exception handling location ein spezieller Step ausgeführt. Dabei wird eine Methode NoRegularNodeAvailable() ausgeführt, die der Programmierer des Agenten bereitstellen muß. In dieser Methode kann der Programmierer dem Auftraggeber der den Agenten gestartet hat, dann irgendwelche Mitteilungen ausgeben (zum Beispiel, keine Flugtickets mehr verfügbar für einen bestimmten Flug) und entsprechend reagieren.

Zu den exception handling locations wird bei der Beschreibung der Stage Construction noch näher eingegangen, die nur dann in einer Stufe existieren werden, wenn nicht genügend regular locations vorhanden sind um eine Stufe bilden.

Abschließend kann gesagt werden, das regular locations reguläre Orte sind auf die der Agent gemäß seiner Reiseroutenspezifikation gehen könnte und Steps laut dieser ausführen. Im Gegensatz dazu wären exception handling locations zwar Orte auf denen man einen Agenten ausführen kann, aber wo man laut Reiseroutenspezifikation nicht hinzugehen hat.

## 5.5 Performance Betrachtungen

In diesem Abschnitt wird auf die Performance des Excatly-Once-Protokolls eingegangen wobei die Ergebnisse der Betrachtung beim Entwurf der Stage Construction berücksichtigt werden.

Um sich die Performance des fehlertoleranten Protokolls zu betrachten, wird ein einfaches Kostenmodell zugrundegelegt ([Straßer 98]). Dabei wird angenommen daß die Kommunikationskosten für eine Nachricht (bestimmter Größe), zwischen den beteiligten Locations (also der Stufe "i" und "i+1"), gleich ist. Aufgrund dieser Annahme kann ausschließlich die Menge der Datentransfers übers Netz betrachtet werden. Eine Möglichkeit die Menge der Datentransfers zu reduzieren, liegt in der Reduzierung der Locations in einer Stufe. Es ist ersichtlich, daß je weniger Locations sich in einer Stufe befinden, desto weniger Datentransfers sind übers Netz durchzuführen während der Migration, dem Monitoring sowie dem Commit. Eine solche Optimierung würde auf der einen Seite dem Ziel des Protokolls, also die Erhöhung der Verfügbarkeit, widersprechen, andererseits hat der Anwendungsentwickler die Möglichkeit, die gesteigerte Verfügbarkeit gegen den Kommunikationsaufwand abzuwägen, indem er entscheidet, wieviele Locations er in einer Stufe benutzt.

Eine weitere Alternative die Menge der Datentransfers übers Netz zu reduzieren, ist die Menge der Daten während der Migration zu reduzieren. Dies kann erreicht werden durch eine sorgfältige Auswahl der Locations zur Bildung einer Stufe, indem der Durchschnitt der Locations der beiden Stufen "i" und "i+1" so groß wie möglich ist, und dadurch die Anzahl der Codeübertragungen des Agenten zu den Locations der anderen Stufe, sich automatisch reduziert. Wenn es in der aktuellen Stufe "i" Locations gibt die sich auch in der Stufe "i+1" befinden, so muß der Code des Agenten während der Migration nicht erneut zu diesen Locations transportiert werden. Wenn zusätzlich noch der Worker der aktuellen Stufe ebenfalls ein Teil der nächsten Stufe ist, zum Beispiel als exception handling location, dann ist



Aus dieser Abbildung geht hervor, dass es zwei I\_Entries gibt, also "3", "5". Bei den restlichen Entries ("1", "2", "4", "6", "7", "8", "9", "10") handelt es sich um B\_Entries. Bei den genannten Einträgen wurden Zahlen, für die Bezeichnung dieser Einträge benutzt.

B\_Entry "1" hat eine höhere Priorität als B\_Entry "2". B\_Entry "2" hat eine höhere Priorität als I\_Entry "3". Innerhalb von I\_Entry "3" befindet sich das I\_Entry "5". Die beiden B\_Entries "7" und "8" befinden sich innerhalb von I\_Entry "5", wobei "7" eine höhere Priorität hat als "8". B\_Entry "4" hat eine höhere Priorität wie I\_Entry "5" und dieses wiederum eine höhere als B\_Entry "6". Weiterhin hat I\_Entry "3" eine höhere Priorität als "9" und dieses wiederum eine höhere als B\_Entry "10". In der jetzt folgenden Betrachtung wird angenommen, daß die Preconditions von "2", "7" und "10" **true** sind, während die restlichen Einträge eine Precondition **false** haben.

### Schritt 1:

Durch die Methode getAllPrecondTrue() ([Buschle 99]) erhält man von der Reiseroute alle B\_Entries deren Precondition true sind. Von Interesse sind ausschließlich die B\_Entries, da ja nur bei diesen die Location und die Methode (Step) angegeben ist, während ein I\_Entry weitere B\_Entries oder auch I\_Entries enthält.

Bei den erhaltenen B\_Entries, handelt es sich um die alternativen Steps, das heißt also die Steps, von denen einer in der nächsten Stufe ausgeführt werden sollte gemäß der Reiseroute, aus der Sicht des Workers, wobei die Menge der Locations im folgenden als NextW bezeichnet werden. Abbildung "5-3a" zeigt die Situation die sich ergibt nach dem Aufruf von getAllPrecondTrue().

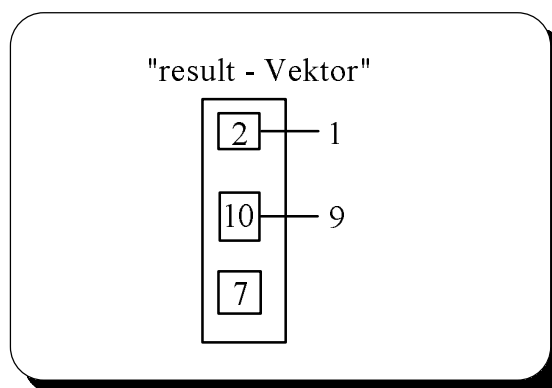


Abbildung 5-3a: "result - Vektor"

Als "result - Vektor" wird der Vektor bezeichnet in der sich die erhaltenen B\_Entries befinden, also "2", "10", "7". Jedes dieser B\_Entries kann eine weitere Liste aus Entries enthalten. In dieser können sich sowohl B\_Entries als auch I\_Entries befinden. Die in der Liste enthaltenen Entries bedeuten, dass sie eine höhere Priorität haben wie das B\_Entry im "result - Vektor". In der Abbildung "5-3a" ist an das B\_Entry "2" im "result - Vektor" eine Liste drangehängt, die das B\_Entry "1" enthält und somit hat "1" eine höhere Priorität als "2". Die Informationen in den Listen, die an den B\_Entries im "result - Vektor" hängen werden von der Reiseroute gelie-

liefert. Da es im Endeffekt eine Totalordnung von den B\_Entries geben muß, müssen eventuelle Prioritäten, die diese erhaltenen B\_Entries in irgendeiner Weise betreffen, berücksichtigt werden. In unserem Beispiel ist momentan zu sehen, daß B\_Entry "1" eine höhere Priorität hat als "2" und "9" eine höhere Priorität als "10". Betrachtet man sich die Abbildung 5-2, dann sieht man, daß B\_Entry "7" sich innerhalb von einem I\_Entry "5" befindet und es gibt ein B\_Entry "4" das eine höhere Priorität als "5" hat, folglich muß auch das B\_Entry "4" eine höhere Priorität haben als das B\_Entry "7". Aus diesem Grund werden alle B\_Entries im "result - Vektor" untersucht, ob sie sich innerhalb eines I\_Entry befinden. Falls ja, muß geprüft werden ob es Entries gibt die eine höhere Priorität haben wie dieser I\_Entry und wenn sich dieser I\_Entry wiederum in einem anderen I\_Entry befindet, muß für diesen wiederum geprüft werden ob es Entries gibt die eine höhere Priorität haben als dieses I\_Entry. Das ganze wird in einer Art rekursiv durchgeführt und die gefundenen Entries die eine höhere Priorität haben somit als das B\_Entry im "result - Vektor" werden in seine Liste hinzugefügt.

In unserem Beispiel hat das B\_Entry "4" eine höhere Priorität wie "5". Daraus folgt das "4" auch eine höhere Priorität hat wie "7" und folglich im "result - Vektor" beim B\_Entry "7" in dessen Liste die die Entries enthält die eine höhere Priorität haben als "7", eingefügt werden. Da das I\_Entry "5" sich in einem anderen I\_Entry "3" befindet muß auch für dieses erneut geprüft werden, ob es Entries gibt die eine höhere Priorität haben als "3". Aus diesen Grund wird auch B\_Entry "2" in die Liste von B\_Entry "7" eingefügt. Abbildung "5-3b" zeigt die Situation die sich nun ergeben hat, nachdem für alle B\_Entries in "result - Vektor" die Prüfung vorgenommen wurde.

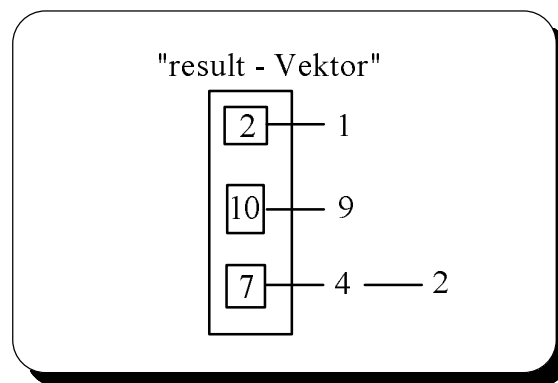


Abbildung 5-3b: "result -Vektor"

### Schritt 2:

In dem Schritt 2 wird die transitive Hülle der Prioritäten berechnet. Dabei wird für jeden B\_Entry im "result - Vektor" seine Liste betrachtet. Jedes Entry in einer Liste kann wiederum eine Liste aus Entries enthalten und so müssen alle Entries in einer Liste (die an einem B\_Entry im "result - Vektor hängt) überprüft werden, ob es weitere Entries gibt die eine höhere Priorität haben als sie und somit auch vom B\_Entry im "result -Vektor".

Als Beispiel wollen wir uns das B\_Entry "9" anschauen, das in der Liste von "10" enthalten ist. In der Liste von "9" ist das I\_Entry "3" enthalten da nach der Reiseroute, das I\_Entry "3" eine

höhere Priorität hat als "9" hat. Das I\_Entry "3" wird somit in die Liste von B\_Entry "10" eingetragen sofern es noch nicht in dieser Liste steht. Als nächstes wird das I\_Entry "3" in der Liste von B\_Entry "10" betrachtet. Das I\_Entry "3" hat in seiner Liste den Eintrag "2", das in die Liste von "10" hinzugefügt wird. Zu beachten ist, das I\_Entry "3" enthält B\_Entries die somit auch eine höhere Priorität haben als das B\_Entry "9" und da "9" eine höhere Priorität als "10" hat, werden die B\_Entries ("7", "8", "4", "6") in die Liste von "10" eingetragen. In der Liste von "10" wird als nächstes das B\_Entry "2" geprüft. Das B\_Entry "1" das eine höhere Priorität als "2" hat, wird in die Liste von "10" eingefügt. In der Reihe der Überprüfungen kommt nun "7" dran. Für das B\_Entry "7", gibt es in der Spezifikation der Reiseroute keine Entries die eine höhere Priorität haben als "7" und somit wird nichts in die Liste von "10" mehr hinzugefügt, bezogen auf diesen Eintrag. In der Liste von Eintrag "8" ist zwar "7" aber dieser befindet sich schon in der Liste "10" und wird nicht eingetragen. Für "4" in der Liste "10" ergibt sich kein weiter Eintrag der hinzugefügt werden muß. Bei Eintrag "6" in der Liste von "10" wird das I\_Entry "5" das eine höhere Priorität wie "6" hat in die Liste von "10" eingetragen. Als nächstes wird der Eintrag "1" überprüft, für den aber sich kein Eintrag in die Liste von "10" ergibt. Damit kommen wir zum I\_Entry "5" in dessen Liste der Eintrag "4" ist. Da aber "4" schon in der Liste von "10" ist, wird kein weiterer Eintrag in "10" eingetragen. Bei "5" handelt es sich um ein I\_Entry, deshalb müßten die B\_Entries innerhalb von "5" auch in die Liste von "10" eingetragen werden. Da sie aber schon in dieser Liste sind werden sie nicht hinzugefügt. Somit wäre die Liste von "10" abgearbeitet. Sind die Listen von B\_Entry "2" und "7" aus "result - Vektor" auch abgearbeitet ergibt sich die Situation in Abbildung "5-3c".

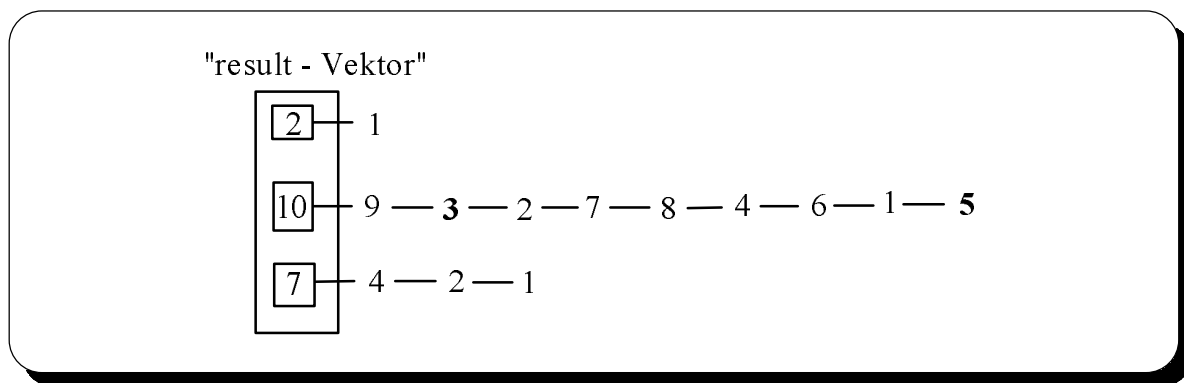


Abbildung 5-3c: "result -Vektor"

### Schritt 3:

Nachdem die transitive Hülle der Prioritäten berechnet ist, wird eine Totalordnung von Prioritäten, zwischen den B\_Entries in "result - Vektor" gebildet. Dabei werden als erstes aus den Listen dieser B\_Entries in "result - Vektor", diejenigen Entries (B\_Entries oder I\_Entries) die nicht im "result - Vektor" enthalten sind entfernt. Dies ist erforderlich da nur eventuelle Prioritäten zwischen diesen B\_Entries benötigt werden. Betrachten wir hierzu als Beispiel die Liste von "7". Dabei sind die B\_Entries "4" und "1" nicht im Vektor enthalten, da ja ihre Precondition **false** ist, und somit werden sie gelöscht aus dieser Liste. Das B\_Entry "2" in die-

ser Liste, ist im "result Vektor" enthalten und bleibt somit erhalten. Falls es in dieser Liste noch ein I\_Entry geben würde, müßte dies gelöscht werden, da nur Prioritäten zwischen B\_Entries in "result - Vektor" von Interesse sind. Dieser Vorgang wird für alle B\_Entries in "result - Vektor" durchgeführt. Abbildung "5-3d" zeigt das Ergebnis von diesem Vorgang.

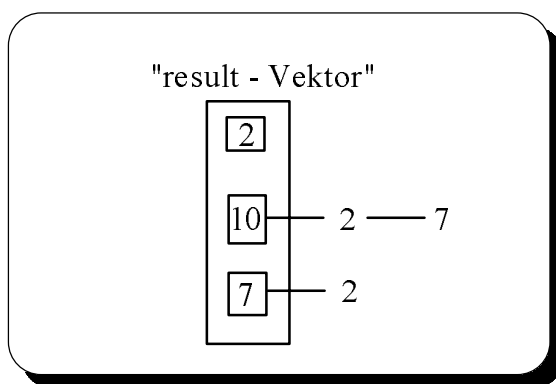


Abbildung 5-3d: "result - Vektor"

Um von diesem Ergebnis zu einer Totalordnung zu kommen muß im Anschluß für jeden B\_Entry in "result - Vektor" geprüft werden, ob dieser in seiner Liste keinen Eintrag hat. Dies würde bedeuten, es gibt kein B\_Entry, das eine höhere Priorität hat als er, und somit wird dieser B\_Entry in den "total - Vektor" aufgenommen. Dieser B\_Entry muß dann aus "result - Vektor" und den restlichen Listen entfernt werden, da dieser B\_Entry abgearbeitet ist. Falls für einen B\_Entry in "result - Vektor" ein Eintrag in seiner Liste vorhanden ist, so wird dieses B\_Entry nicht weiter behandelt und der nachfolgende B\_Entry in "total - Vektor" betrachtet. In unserem Beispiel hat "2" niemanden in seiner Liste und wird deshalb in "total - Vektor" eingetragen. Dabei muß "2" aus den restlichen Listen und dem "result - Vektor" entfernt werden. Es ergibt sich folgende Situation die in Abbildung "5-3e" aufgezeigt wird.

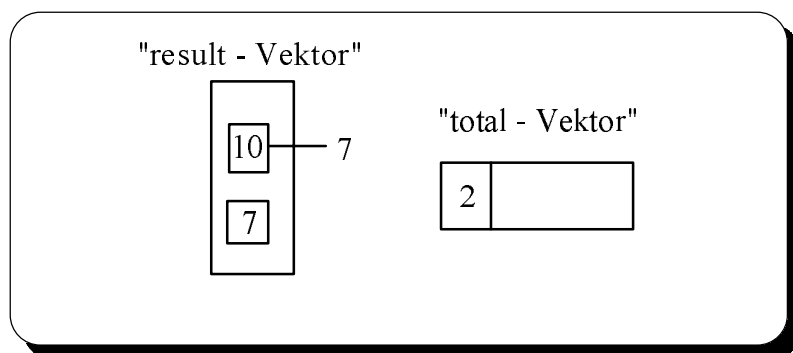


Abbildung 5-3e: "result - Vektor" und "total - Vektor"

Da als nächstes der Eintrag "10" geprüft wird, dieser aber einen Eintrag in seiner Liste enthält kann dieser nicht in "total - Vektor" hinzugefügt werden, da es einen Eintrag gibt der eine höhere Priorität hat wie dieser. Beim B\_Entry "7" ist kein Eintrag in seiner Liste und deshalb

wird dieser in den "total - Vektor" eingefügt und aus "result - Vektor" und den restlichen Listen entfernt. Abbildung "5-3f" verdeutlicht dies.

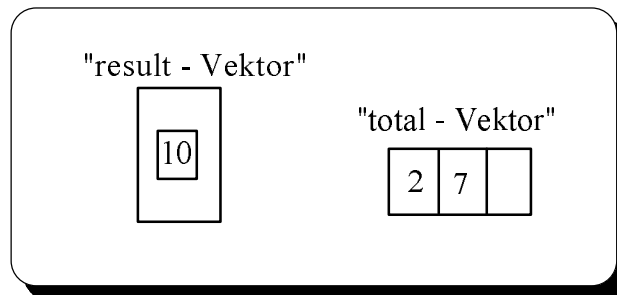


Abbildung 5-3f: "result - Vektor" und "total - Vektor"

Da "result - Vektor" nun aus nur einen B\_Entry besteht ("10"), das keinen Eintrag in seiner Liste hat wird, auch B\_Entry "10" in den "total - Vektor" eingefügt und aus "result - Vektor" entfernt. Das Ergebnis aus dem Schritt 3 ist ein "total - Vektor" mit B\_Entries, bei denen die Prioritäten zwischen den B\_Entries, falls für diese welche definiert wurden, berücksichtigt werden und somit eine Totalordnung von B\_Entries (Locations) erzeugt wurde. In Abbildung "5-3g" wird das Ergebnis nach Vervollendung von Schritt 3 dargestellt.

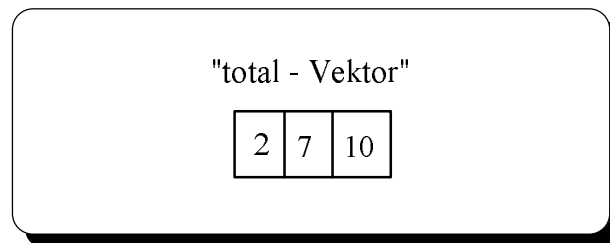


Abbildung 5-3g: "total - Vektor"

#### Schritt 4:

Die im "total - Vektor" enthaltenen B\_Entries (NextW) müssen als nächstes auf Verfügbarkeit geprüft werden, da es nicht sinnvoll ist, die nächste Stufe aus Locations zu bilden, die nicht verfügbar sind. Weiterhin erfolgt innerhalb von NextW eine Prüfung, ob es B\_Entries gibt die den gleichen Location - Namen haben. Falls es in NextW zwei B\_Entries gibt die den gleichen Location - Namen haben, würde dies bedeuten, das bei einer Location zwei alternative Steps ausgeführt werden können. Sinnvollerweise wird das B\_Entry mit der niedrigeren Priorität aus NextW gelöscht. Wir betrachten hierzu als Beispiel, ein NextW mit den B\_Entries "A", "B", "C" (Abbildung "5-4"). Nehmen wir an die beiden B\_Entries "A" und "C" haben den gleichen Location - Namen. Da "A" weiter vorne im "NextW - Vektor" ist wird es eine höhere Priorität haben als "C" und somit wird "C" aus NextW gelöscht.

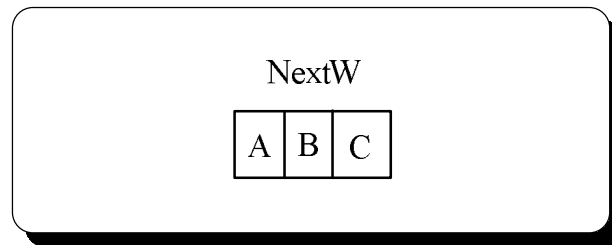


Abbildung 5-4: NextW

Die Anzahl der Locations aus der die nächste Stufe bestehen soll wird mit "N" bezeichnet. Abhängig von der Anzahl der B\_Entries in NextW und der Anzahl der Locations aus der die nächste Stufe bestehen soll, werden 3 Fälle unterschieden:

**Fall 1:**

Wenn die Anzahl der B\_Entries in **NextW** gleich ist mit der Anzahl der Locations aus der die nächste Stufe "i+1" bestehen soll ("N", per Defaultwert wird die nächste Stufe aus 3 Locations bestehen, diese Anzahl kann aber geändert werden), so bilden die Einträge in NextW die nächste Stufe "i+1", wobei das erste Element im "NextW - Vektor", der B\_Entry mit der höchsten Priorität ist innerhalb von NextW, das zweite Element der B\_Entry mit der zweithöchsten Priorität ist, und so weiter. Das heißt, daß der erste B\_Entry den Worker repräsentieren wird, während die restlichen B\_Entries die Observer - Rolle in der nächsten Stufe annehmen. Die nächste Stufe wird somit in diesem Fall nur aus "regular locations" bestehen.

Die Reiseroute kann Prioritätsfestlegungen vornehmen. Es sind aber keine absoluten Prioritäten. Im Protokoll allerdings muß es eine Totalordnung der Prioritäten zwischen den Locations geben. Nur so kann eine Location die die höchste Priorität hat zum Worker ernannt werden. und beim Ausfall dieses Workers wird dann die Location mit der zweithöchsten Priorität die Rolle des Workers annehmen.

**Fall 2:**

Wenn die Anzahl der B\_Entries in NextW größer ist als "N", wird zur Bildung der nächsten Stufe "i+1" (" $S_{i+1}$ ") aus Performance Gründen (Abschnitt 5.5) der Durchschnitt zwischen der aktuellen Stufe (" $S_i$ ") und NextW gebildet. Es werden dabei zwei Fälle unterschieden:

**Fall 2a:**

Ist die Anzahl  $|S_{i+1}| \geq N$ , so werden "N" B\_Entries aus NextW, die auch in der aktuellen Stufe "i" enthalten sind, gemäß ihrer Priorität die nächste Stufe "i+1" bilden. Betrachten wir hierzu Abbildung "5-5". Die Stufe "i" besteht aus den 5 B\_Entries ("A", "B", "C", "D", "E") und NextW aus 9 B\_Entries ("G", "E", "I", "B", "A", "X", "C", "Y", "D", "Z"). Im "endTotal - Vektor" befinden sich der Durchschnitt der Stufe "i" und NextW wobei dieser gleich "N=5"

ist. Die B\_Entries im "endTotal - Vektor" werden die nächste Stufe "i+1" bilden. Die nächste Stufe "i+1" wird also aus "regular locations" bestehen.

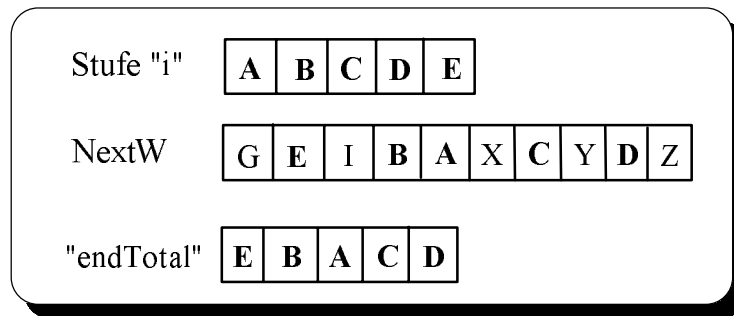


Abbildung 5-5: Fall 2a

Wie in dieser Abbildung zu sehen ist, hat in NextW "G" eine höhere Priorität als "E", "I" eine höhere als "B", "X" eine höhere wie "C" und "Y" eine höhere als "D". In dem "endTotal - Vektor" aber wurden diese B\_Entries ("G", "I", "X", "Y") nicht berücksichtigt, da aus Gründen der Performance und gegen die Prioritäten entschieden wurde.

### Fall 2b:

Ist die Anzahl  $|S_{i+1}| < "N"$ , so werden die restlichen B\_Entries  $m = "N" - |S_{i+1}|$  aus  $"NextW \setminus S_i"$ , gemäß ihrer Priorität genommen. Abbildung 5-6 verdeutlicht dies. Die aktuelle Stufe besteht aus 5 B\_Entries ("G", "A", "I", "B", "F") und NextW aus 9 B\_Entries ("X", "Y", "B", "K", "L", "M", "A", "O", "P", "H"). Weiterhin soll  $"N = 5"$  sein. In "endTotal" befinden sich nun die B\_Entries aus dem Durchschnitt von NextW und der Stufe "i" (also "A", "B") sowie 3 weitere B\_Entries aus NextW. Die Priorität die diese B\_Entries in NextW haben, wird dabei berücksichtigt. In diesem Fall würde die nächste Stufe aus "regular locations" bestehen.

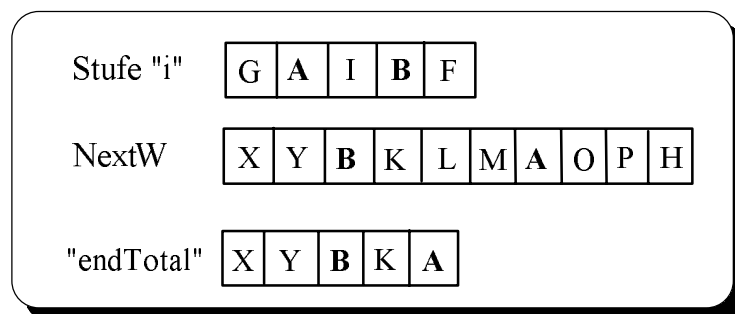


Abbildung 5-6: Fall 2b

### Fall 3:

Wenn die Anzahl der B\_Entries in NextW kleiner als "N" ist, werden die restlichen B\_Entries

die benötigt werden, also  $m = N - |\text{NextW}|$  von der aktuellen Stufe "i" geholt. Diese "m" B\_Entries müssen verfügbar sein, und ihr Location - Name darf nicht in NextW enthalten sein. Bei diesen "m" B\_Entries handelt es sich um "exception handling locations", die am Ende von den "regular location" (sind die am Anfang erhaltenen B\_Entries in NextW, nachdem überprüft wurde ob es in NextW gleiche Location - Namen gibt) im "endTotal - Vektor" hinzugefügt werden, da sie eine niedrigere Priorität haben müssen als die "regular locations". Falls man aus der aktuellen Stufe "i" nicht genügend "m" B\_Entries erhält so wird aus einem Konfigurationsfile die restlichen Einträge geholt. Es wird angenommen das eine Location noch weitere Locations kennt. Von diesen Locations werden die Location - Namen in dieses Konfigurationsfile abgelegt und dienen dazu, falls aus der Stufe "i" nicht genügend B\_Entries zur Bildung einer Stufe ausreichen, dann die Locations im Konfigurationsfile als "exception handling locations" zu verwenden. Bei diesen "m" B\_Entries handelt es sich um "exception handling locations", da sie nicht in NextW (die gemäß der Reiseroute, alternativ ausgeführt werden können) sein werden. Die Locations im Konfigurationsfile müssen wiederum verfügbar sein, und ihr Location - Name darf nicht im "endTotal - Vektor" enthalten sein.

Für den Fall das weder aus der aktuellen Stufe noch aus dem Konfigurationsfile die benötigten "m" B\_Entries zu erhalten sind, weil sie nicht verfügbar oder ihr Location - Name vielleicht schon in NextW ist, werden die B\_Entries aus der Stufe "i" und dem Konfigurationsfile die nicht verfügbar sind und ihr Location - Name sich nicht im "endTotal - Vektor" befindet, solange geprüft, bis wieder welche verfügbar sind und man die restliche Anzahl "m" erhalten hat.

Der Fall 3 soll anhand eines Beispiel verdeutlicht werden. In Abbildung 5-7 besteht die aktuelle Stufe "i" aus 5 B\_Entries ("A", "H", "K", "D", "F"), NextW aus 3 B\_Entries ("D", "B", "F") und die nächste Stufe soll zum Beispiel aus  $N = 5$  Locations bestehen. Die B\_Entries in NextW werden im "endTotal - Vektor" übernommen und die restlichen  $m=2$  B\_Entries, aus der aktuellen Stufe "i". Vorausgesetzt, das die B\_Entries "A" und "H" verfügbar sind und ihre Location nicht im "endTotal - Vektor" enthalten ist, werden sie als "exception handling locations" in dem "endTotal - Vektor" übernommen. Dabei war "A" in der Stufe "i" der Worker und ist nun ein Teil der neuen Stufe, was aus Performance Betrachtungen (Abschnitt 5.5) vorteilhaft ist. Die nächste Stufe wird somit aus "regular locations" und "exception handling locations" bestehen.

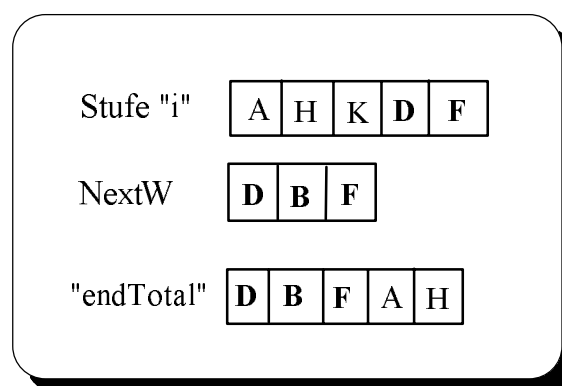


Abbildung 5-7: Fall 3

## 5.7 Entwurf der Stage - Construction

Im folgenden Abschnitt wird auf den Entwurf der Stage - Construction näher eingegangen. Abbildung 5-8 gibt einen Überblick über die Teile der Mole - API die beim Entwurf der Stage - Construction verändert werden, dabei ist folgende Notation zu beachten:

**V** = Veränderung der Klasse oder der Methode.

**N** = Erzeugung der Methode oder der Variablen.

Eine Ausführliche Beschreibung dieser Klassen und Methoden vom Agentensystem Mole 3.0, sowie die Methoden und Variablen die zusätzlich noch hinzugefügt wurden, sind im Anhang "A.2" aufgeführt.

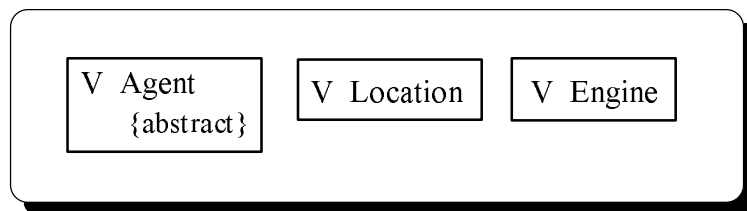


Abbildung 5-8: Überblick des Entwurfs Stage - Construction

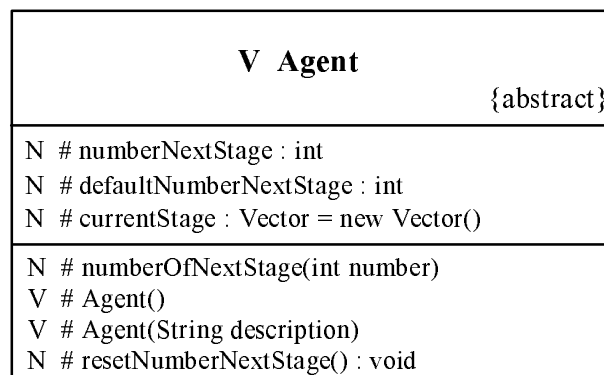


Abbildung 5-9: Agent - Klasse

### "currentStage"

Nachdem der Algorithmus (zur Bildung der nächsten Stufe), die B\_Entries für die nächste Stufe bestimmt, auf der ein Agent durch eine Put - Operation geschickt wird, werden diese B\_Entries in "currentStage" abgelegt. Diese werden benötigt, wenn zur Bildung der nächsten Stufe aus Performance Gründen der Durchschnitt zwischen der aktuellen Stufe "i" und NextW gebildet wird. Weiterhin, falls aus NextW die B\_Entries nicht ausreichen sollten um die nächste Stufe zu bilden und so B\_Entries aus "currentStage" verwendet werden (Performance

Grund). Außerdem werden sie benötigt, weil jede Location in einer Stufe die "currentStage" kennen muß.

### "numberOfNextStage(int number)"

Dem Programmierer eines Agenten wird die Möglichkeit zur Verfügung gestellt Einfluß zu nehmen auf die Anzahl der Locations aus der die nächste Stufe bestehen soll. Dadurch kann die Zuverlässigkeit einer Stufe, daß der Agent ausgeführt und nicht blockiert wird, beeinflußt werden.

Im Gegensatz dazu kann beim Start einen Agenten (von der Console), unter Angabe des Tag "Stage" und einer Zahl, für alle Stufen die dieser Agent besuchen wird, die Anzahl der Locations aus der die Stufen bestehen sollen, festgelegt werden.

Beispiel:

```
mole.apps.Trader(Stage 7) at testlocation4
```

Alle Stufen die der Agent besuchen wird, werden aus 7 Locations bestehen. Der Programmierer kann auch beide Möglichkeiten (Bestimmung der Anzahl der Locations einer Stufe) benutzen.

Zum Beispiel wird "Stage 7" und vielleicht in einer Stufe "i" innerhalb eines Steps auch noch die Methode "numberOfNextStage(5) benutzt, dabei wird dann die nächste Stufe aus 5 Locations bestehen, während die restlichen aus 7 bestehen. Vor der Stufe "i" werden die Stufen aus 7 Locations bestehen.

Wie schon erläutert muß eine Stufe aus einer ungeraden Anzahl von Locations bestehen. Wenn eine gerade Zahl bei der Verwendung der Methode "numerOfNextStage()" oder dem Tag "Stage" eingegeben, so wird die nächst kleinere ungerade Zahl genommen um die Anzahl der Locations für die nächste Stufe oder für alle Stufen zu bestimmen.

<b>V Location</b>
N \$ - globalDefaultNumberNextStage : int = 3 N # result : Vektor N # endTotal : Vektor = new Vector()
N \$ # getGlobalDefaultNumberNextStage() : int V - _arrive(...) : boolean N - startAgent(...) : void N # stageConstruction(Agent anAgent) : void N - B_EntryLocationName() N - notInVector(Vector vector, LocationName locName) : boolean N - indexOfNull(Vector vector, int index) : int

Abbildung 5-10: Location - Klasse

**"globalDefaultNumberNextStage"**

Die Stufen werden, soweit nicht beim Start eines Agenten oder die Methode "numberOfNextStage()" benutzt wird, aus 3 Locations bestehen. Wird in einem Step die Methode "numberOfNextStage(5)" verwendet, so wird die nächste Stufe aus 5 Locations bestehen, während die restlichen aus 3 Locations.

**"startAgent"**

Zum Zeitpunkt wo der Agent von der Console gestartet wird, existiert noch keine Stufe für diesen Agenten. Wenn aber der Stage - Construction Algorithmus verwendet wird, um die nächste Stufe "i+1" zu bilden, muß eine aktuelle Stufe schon vorhanden sein, damit falls notwendig die B\_Entries von der aktuellen Stufe bei der Bildung der nächsten Stufe benutzt werden. Deshalb wird, wenn der Agent gestartet wird eine erste Stufe gebildet aus B\_Entries, wobei jeder B\_Entry einen Location - Namen haben wird, der einen Namen aus dem Konfigurationsfile "knownLocations.cfg" entspricht. Weiterhin wird dem B\_Entry die Methode "start" hinzugefügt. Diese B\_Entries werden in "currentStage" abgelegt die vom Exactly-Once-Protokoll verwendet werden und die Methode "start" ausgeführt. Die Methode "start" eines Agenten, wird die Reiseroutenspezifikation eines Agenten enthalten und nur ein einziges mal in der ersten Stufe ausgeführt werden, während in den übrigen Stufen die der Agent besucht die Methoden die die Reiseroutenspezifikation vorgibt.

**"stageConstruction(Agent anAgent)"**

Mit der Methode "stageConstruction()" werden die B\_Entries (Locations) für die nächste Stufe bestimmt (siehe Abschnitt 5.6).

<b>V Engine</b>
N knowLoc : Vector = new Vector()
N - readConfig() : void
V - init (...) : void
V \$ + startNewAgent(...) : boolean

Abbildung 5-11: Engine - Klasse

**"readConfig()"**

Es wird angenommen, das eine Location noch weitere Locations kennt. Die Location - Namen müssen in "knownLocations.cfg" reingeschrieben werden, von demjenigen der die location kontaktiert. In einem System mit zum Beispiel 10 Rechner die alle Zugriff haben auf das selbe File - System, wird es auf jedem Rechner nur eine Engine und eine Location (diese Festlegung wird in Kapitel 6 näher erläutert) existieren. Jede dieser Engines braucht die Information welche Locations sie noch kennt, und wird auf das Konfigurationsfile "knownLocations.cfg"

zugreifen. Der Programmierer muß für alle Engines (auf verschiedenen Rechnern) die er starten will, zuerst die Location - Namen ins "knownLocations.cfg" reinschreiben. Erst dann wird er bei einer Engine einen Agenten starten.

Die Location - Namen werden mit der Methode "readConfig()" aus dem Konfigurationsfile "knownLocations.cfg" gelesen, und in "knowLoc" abgelegt. Für jede einzelne Engine steht in diesem Konfigurationsfile einmal sie selber drinnen, deshalb wird beim Lesen (readConfig()) die Prüfung vorgenommen, ob von der gestarteten Engine die Location in diesem Konfigurationsfile ist, falls ja dann wird sie nicht in "knownLoc" aufgenommen. In der Datei "locations.dat" muß Mole mitgeteilt werden welche Locations existieren unter Angabe der IP-Adresse der Engine, der TCP - Portnummer und der Location (Beispiel 192.168.2.1:9876 location1). In dem Konfigurationsfile bei dem die Engine gestartet wird, muß zusätzlich noch folgender Eintrag stehen:

**ENGINE.POSSIBLESTAGELOCATIONS knownLocations.cfg**

## 5.8 Anforderungen an die Reiseroutenspezifikation

Aus dem Entwurf der Stage Construction ergeben sich für die Studienarbeit ([Buschle 99]) folgende Anforderungen:

- a) Welche alternativen Steps (Schritte) sind aus der Sicht des Workers (Stufe "i") in der nächsten Stufe "i+1" ausführbar.
- b) Informationen über Prioritätsfestlegungen zu liefern, zwischen den Entries wenn vorhanden. Zum Beispiel der B\_Entry "X" hat eine höhere Priorität als B\_Entry "Y".

## 6. Integration des Exactly - Once - Protokolls in Mole

### 6.1 Vorbemerkungen

Das Excatly-Once-Protokoll soll nachdem Änderungen in der Mole - API vorgenommen wurden, in die eingeschränkte Mole Version integriert werden. In der momentanen Version des Protokolls ist es so, daß wenn etwas migriert, migriert kein Agent sondern Bytes im Exactly-Once-Protokolls. Dies wird nun soweit geändert, daß nun Agenten migrieren. Der Agent wird aus der Queue geholt, verarbeitet und zu den Queues der nächsten Stufe geschickt. Es wird also ein Mechanismus benötigt, der den Agenten aus der Warteschlange herausholt, und den Agenten ausführt und ihm dann zu den Zielqueues der nächsten Stufe weiterschickt. Somit sind auch auf der Seite des Protokolls Anpassungen erforderlich.

### 6.2 Möglichkeiten der Integration

Wenn auf einem Rechner eine Engine und mehrere Locations existieren würden, und die nächste Stufe gebildet wird mit genau diesen Locations, so wäre dies ungünstig. Der Nachteil ist, beim Ausfall dieses Rechners könnte der Agent nicht in die nächste Stufe gelangen, da alle Locations von diesem Rechner nicht mehr erreichbar sind um einen Step des Agenten auszuführen. Weiterhin kann sich auch folgende Situation ergeben: Der Rechner fällt aus, während der Agent in einer Stufe ist, dabei fallen gleich mehrere Location aus.

Es gibt zwei Möglichkeiten um sicherzustellen, daß in einer Stufe nicht zwei Locations auf dem selben Rechner sind:

**Fall 1:**

Wenn auf einem Rechner eine Engine und genau eine Location existieren, so ergibt sich der soeben erwähnte Nachteil nicht.

**Fall 2:**

Bei der Möglichkeit, daß auf einem Rechner mehrere Engines und mehrere Locations existieren, muß sichergestellt werden, daß wenn die nächste Stufe gebildet wird, geprüft wird ob die Locations auf unterschiedlichen Rechnern sind, damit der obige Nachteil nicht auftritt. Für einen Location - Namen erhält man die IP - Adresse und die Port - Nummer. Bei unterschiedlichen IP - Adressen wird es sich meistens um unterschiedliche Rechner handeln. Bei gleichen IP - Adressen aber unterschiedlichen Ports sind es unterschiedliche Engines.

Um die prototypische Implementierung möglichst einfach zu halten wird festgelegt, daß auf einem Rechner nur eine Engine mit genau einer Location existiert (das heißt Fall 1). Aufgrund dieser Festlegung ist es dann nicht möglich, bei der Bildung der nächsten Stufe zwei Locations auf einem Rechner zu haben.

## 6.3 Entwurf

Die Worker - Klasse muß insoweit geändert werden, daß der Agent aus der Queue geholt wird, auf der Location des Agenten den Step ausführt, sowie nach der Abarbeitung des Steps den Agenten durch die Put - Operation innerhalb der Transaktion in die Queues der nächsten Stufe ablegt. Durch die Ausführung der executeAgent() - Methode von der Location Klasse wird der Step des Agent ausgeführt. Nur wenn der Step erfolgreich ausgeführt wird, wird der Agent anschließend in die nächste Stufe übergeben, im anderen Fall wird die Transaktion abgebrochen.

Von der Engine - Klasse werden die Objekte des Protokolls (Timer, Administrator, Voter, Coordinator\_Observer, Orchestrator) gestartet. Abbildung 6-1 verdeutlicht das Verarbeitungsmodell.

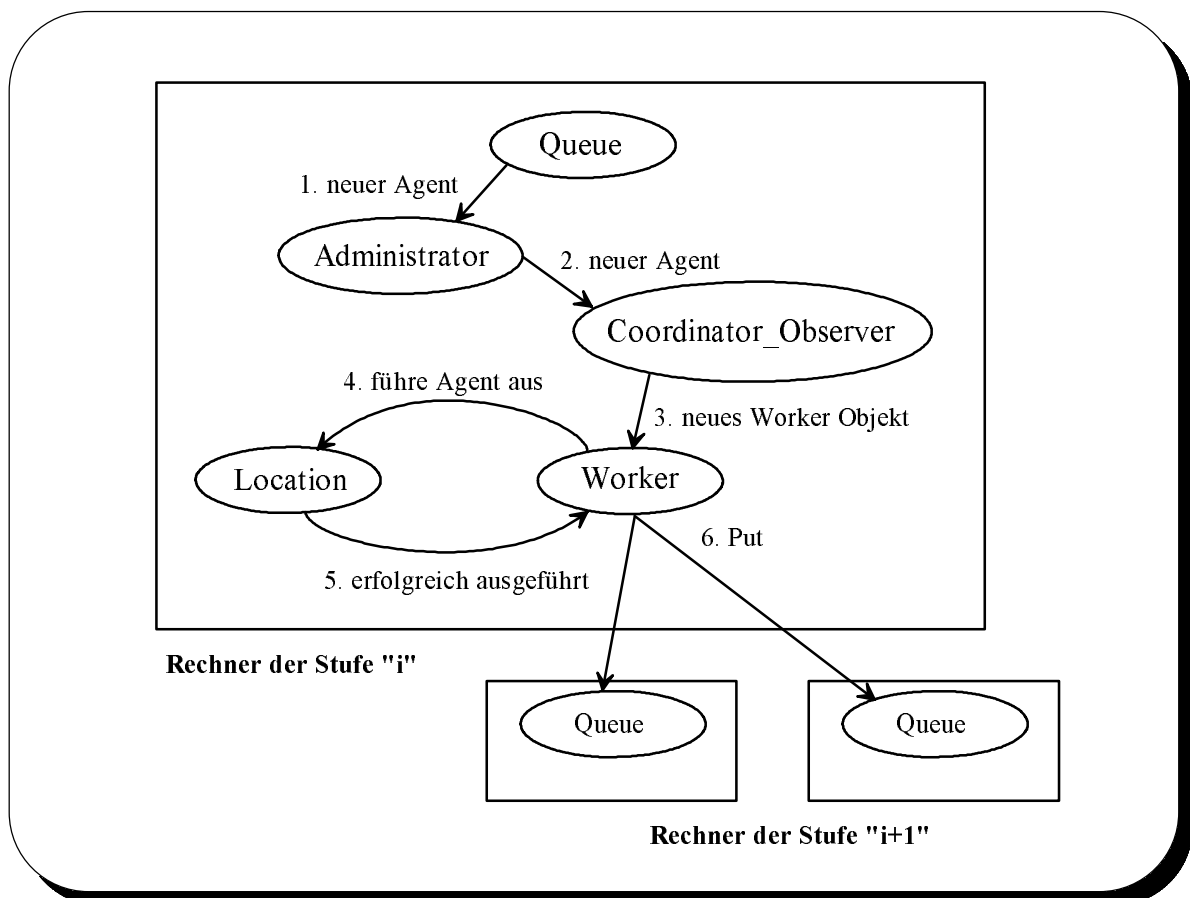


Abbildung 6-1: Verarbeitungsmodell

Dem Administrator wird mitgeteilt, das ein Agent in der Queue ist. Dieser wiederum informiert den Coordinator\_Observer. Der Coordinator startet sofort den Worker und dieser

teilt der Location mit, den Agenten auszuführen. Wenn die Location den Agenten erfolgreich ausgeführt hat meldet sie dies dem Worker. Dieser legt dann den Agenten zu den Queues der nächsten Stufe ab.

Dieses Verarbeitungsmodell in Abbildung 6-1 ist die reine Ausführung vom Agenten ohne das jetzt noch zusätzlich das Voting Protokoll mit dargestellt worden ist.

## 7 Zusammenfassung und Ausblick

In diesem Kapitel werden die wichtigsten Punkte noch einmal zusammengefaßt und es wird ein Ausblick auf mögliche Änderungs- und Erweiterungsmöglichkeiten gegeben.

### 7.1 Zusammenfassung

Im Fokus dieser Arbeit lagen drei Aspekte:

Um das Protokoll in Mole sinnvoll zu integrieren war es erforderlich Änderungen vorzunehmen die sowohl die Kommunikationspartner, Kommunikationsmechanismen, die Migration, sowie die Erzeugung von weiteren Agenten betroffen hatte.

Eine grundlegende Funktionalität (Stage Construction) zur Bestimmung des Workers und der Observer für die nächste Stufe, die das Exactly-Once-Protokoll benötigt, wurde in dieser Arbeit entworfen und implementiert.

Die Integration des Protokolls in Mole wurde in Kapitel 6 beschrieben.

### 7.2 Ausblick

Im folgenden werden Erweiterungs- und Änderungsmöglichkeiten vorgestellt.

Es kann davon ausgegangen werden, daß nicht sehr viele Leute einen Corba Transaction Service verwenden. Deshalb wäre es sinnvoll wenn man eine Mole - Version hat, bei der sowohl transaktionale als auch nicht transaktionale Agenten benutzt werden können. Transaktionale Agenten können nur mit bestimmten Agenten kommunizieren während nicht transaktionale Agenten mit jedem beliebigen Agenten kommunizieren können. In der Spezialversion von Mole die in dieser Arbeit erarbeitet wurde, werden ausschließlich Agenten verwendet die nach diesem Protokoll arbeiten (transaktionale Agenten).

Es wäre durchaus denkbar, daß ein Agent in einem Step einen anderen Agenten erzeugen kann. Sicherergestellt werden muß, das der erzeugte Agent tatsächlich nur dann aktiv wird wenn nachdem der Step erfolgreich ausgeführt wurde. Wenn ein Agent neu eingebracht wird, wird für ihn eine Stufe gebildet und durch die Put - Operation wird der Agent innerhalb einer Transaktion zu den Queues der Locations der ersten Stufe gebracht. Wenn man nun einen Agenten innerhalb eines Steps erzeugen möchte so muß prinzipiell dasselbe gemacht werden, wie wenn ein Agent eingebracht wird, das heißt es wird eine neue Stufe gebildet und der erzeugte Agent wird dort abgelegt. Dies wird aber jetzt nicht in einer eigenen Transaktion gemacht, sondern müßte innerhalb der Transaktion gemacht werden die den Step ausführt.

Dies würde bedeuten der erzeugte Agent würde nicht aktiv werden, sondern müßte auch zuerst in die Eingangsqueue seiner ersten Stufe gebracht werden und zwar innerhalb der Transaktion wo der Step ausgeführt wird. Dann wäre es garantiert, daß wenn der Step erfolgreich beendet wird, dann auch der erzeugte Agent in seiner ersten Stufe ist und daß wenn die Transaktion abgebrochen wird, daß dann der erzeugte Agent auch verschwindet. Somit wäre sichergestellt das der erzeugte Agent nur dann aktiv wird wenn tatsächlich auch die Stufe erfolgreich beendet ist.

Eine weitere Erweiterung die vorstellbar wäre, wenn es Mechanismen zum partiellen zurücksetzen (Rollback) der Agentenausführung gebe. Wenn man einen oder mehrere Steps ausgeführt hat soll es also die Möglichkeit geben den Step oder mehrere Steps zurücksetzen zu können.

## Anhang "A"

### Anhang "A.1"

Im Anhang "A.1" befinden sich eine Ausführliche Beschreibung der im Kapitel 4 aufgeführten Klassen, Schnittstelle, Methoden, Variablen vom Agentensystem Mole 3.0, sowie auch Klassen, Variablen die zusätzlich hinzugefügt wurden.

#### Class Agent

```
/**
 * The father of all agents. The root in of the class hierarchy from which
 * all agents must inherit.
 */
public abstract class Agent
    extends Object
    implements Serializable

/**
 * a list to store incoming messages in
 */
protected Vector messageList = new Vector() // The agent's message list

protected Coordinator context // contains the transaction context

/**
 * called to dispatch an incoming message to the appropriate
 * receive method
 *
 * @param m the message to be dispatched
 */
protected void dispatchMessage(Message m)

/**
 * started from method dispatchMessage
 * to handle a Message
 *
 * @param m the message to be received
 */
public void receiveMessage(Message m)
```

```
/**
 * sends a message asynchronously
 */
public final void message(Message m)

/**
 * sends a message with errorsemantics 0
 *
 * @param receiver the name of the receiver of the message
 * @param receiverLocation the name of the receiver's location
 * @param mid the message id
 * @param o the content of the message
 */
public void sendUnreliable(AgentName receiver,
                          LocationName receiverLocation, long mid, Serializable o)

/**
 * sends a message with errorsemantics 1
 *
 * @param receiver the name of the receiver of the message
 * @param receiverLocation the name of the receiver's location
 * @param mid the message id
 * @param o the content of the message
 */
public void sendReliable(AgentName receiver, LocationName receiverLocation,
                          long mid, Serializable o)

/**
 * sends a message with errorsemantics 2
 *
 * @param receiver the name of the receiver of the message
 * @param receiverLocation the name of the receiver's location
 * @param mid the message id
 * @param o the content of the message
 */
public void sendMailbox(AgentName receiver, LocationName receiverLocation,
                          long mid, Serializable o)

/**
 * calls a remote method with no parameters
 *
 * @param procName the name of the remote procedure
 * @param remoteAgent an agent that provides the called method
 * @param remoteLocation the destination location
 * @return the serializable object returned by the remote method
 */
```

```
public final Serializable call(String procName, PeerQualifier remoteAgent,  
                                LocationName remoteLocation)  
    throws NoSuchReceiverException, NoSuchLocationException,  
           NoSuchMethodException, RemoteException, UnknownHostException,  
           NotBoundException, IllegalAccessException, InvocationTargetException
```

```
/**  
 * calls a remote method with one parameter  
 *  
 * @param procName the name of the remote procedure  
 * @param remoteAgent an agent that provides the called method  
 * @param remoteLocation the destination location  
 * @param p1 the parameter for the remote method  
 * @return the serializable object returned by the remote method  
 */
```

```
public final Serializable call(String procName, PeerQualifier remoteAgent,  
                                LocationName remoteLocation, Serializable p1)  
    throws NoSuchReceiverException, NoSuchLocationException,  
           NoSuchMethodException, RemoteException, UnknownHostException,  
           NotBoundException, IllegalAccessException, InvocationTargetException
```

```
/**  
 * calls a remote method with two parameters  
 *  
 * @param procName the name of the remote procedure  
 * @param remoteAgent an agent that provides the called method  
 * @param remoteLocation the destination location  
 * @param p1,p2 the parameters for the remote method  
 * @return the serializable object returned by the remote method  
 */
```

```
public final Serializable call(String procName, PeerQualifier remoteAgent,  
                                LocationName remoteLocation, Serializable p1,  
                                Serializable p2)  
    throws NoSuchReceiverException, NoSuchLocationException,  
           NoSuchMethodException, RemoteException, UnknownHostException,  
           NotBoundException, IllegalAccessException, InvocationTargetException
```

```
/**  
 * calls a remote method with three parameters  
 *  
 * @param procName the name of the remote procedure  
 * @param remoteAgent an agent that provides the called method  
 * @param remoteLocation the destination location  
 * @param p1,p2,p3 the parameters for the remote method  
 * @return the serializable object returned by the remote method  
 */
```

```
public final Serializable call(String procName, PeerQualifier remoteAgent,  
                                LocationName remoteLocation, Serializable p1,  
                                Serializable p2, Serializable p3)  
    throws NoSuchReceiverException, NoSuchLocationException,  
           NoSuchMethodException, RemoteException, UnknownHostException,  
           NotBoundException, IllegalAccessException, InvocationTargetException
```

```
/**  
 * calls a remote method with four parameters  
 *  
 * @param procName the name of the remote procedure  
 * @param remoteAgent an agent that provides the called method  
 * @param remoteLocation the destination location  
 * @param p1,p2,p3,p4 the parameters for the remote method  
 * @return the serializable object returned by the remote method  
 */
```

```
public final Serializable call(String procName, PeerQualifier remoteAgent,  
                                LocationName remoteLocation, Serializable p1,  
                                Serializable p2, Serializable p3, Serializable p4)  
    throws NoSuchReceiverException, NoSuchLocationException,  
           NoSuchMethodException, RemoteException, UnknownHostException,  
           NotBoundException, IllegalAccessException, InvocationTargetException
```

```
/**  
 * calls a remote method with five parameters  
 *  
 * @param procName the name of the remote procedure  
 * @param remoteAgent an agent that provides the called method  
 * @param remoteLocation the destination location  
 * @param p1,p2,p3,p4,p5 the parameters for the remote method  
 * @return the serializable object returned by the remote method  
 */
```

```
public final Serializable call(String procName, PeerQualifier remoteAgent,  
                                LocationName remoteLocation, Serializable p1,  
                                Serializable p2, Serializable p3, Serializable p4,  
                                Serializable p5)  
    throws NoSuchReceiverException, NoSuchLocationException,  
           NoSuchMethodException, RemoteException, UnknownHostException,  
           NotBoundException, IllegalAccessException, InvocationTargetException
```

```
/**  
 * calls a remote method with an unspecified number of parameters  
 *  
 * @param procName the name of the remote procedure  
 * @param remoteAgent an agent that provides the called method  
 * @param remoteLocation the destination location
```

```

* @param param an array with the parameters for the remote method
* @exception NoSuchReceiverException, if the remote agent isn't found
* @exception NoSuchLocationException, if the remote location isn't found
* @exception NoSuchMethodException, if no suitable method is found
* @exception RemoteException, if the registry couldn't be contacted
* @exception UnknownHostException, if the destination host isn't found
* @exception NotBoundException, if the destination location's dispatchRPC
* method isn't bound to the registry
* @exception IllegalAccessException, if there were problems accessing
* items on the remote engine
* @exception InvocationTargetException, if the called method has thrown
* an exception during execution
* @return the serializable object returned by the remote method
*/
public final Serializable call(String procName, PeerQualifier remoteAgent,
    LocationName remoteLocation, Serializable[] param)
    throws NoSuchReceiverException, NoSuchLocationException,
    NoSuchMethodException, RemoteException, UnknownHostException,
    NotBoundException, IllegalAccessException, InvocationTargetException

/**
 * create a new, unrelated agent with no additional parameters,
 * @return the name assigned to the agent, null if an error occurred
 */
final public AgentName createAgent(Agent newAgent)

/**
 * create a new unrelated agent with parameters.
 * @param parameters parameter list
 * @return the name assigned to the agent, null if an error occurred
 */
final public AgentName createAgent(Agent newAgent, Hashtable parameters)

/**
 * create a child agent with no additional parameters,
 * @return the name assigned to the agent, null if an error occurred
 */
final public AgentName createChild(Agent newAgent)

/**
 * create child with parameters,
 * @param parameters parameter list
 * @return the name assigned to the agent, null if an error occurred
 */
final public AgentName createChild(Agent newAgent, Hashtable parameters)

```

---

```
/**
 * called to unregister a heartbeat agent temporary
 * @see Periodical
 */
protected final void HeartbeatOff()

/**
 * called to register a heartbeat agent temporary
 * @see Periodical
 */
protected final void HeartbeatOn()

/*****
/** Heartbeat          */
private class HeartbeatAgentThread
    extends MoleThread

/**
 * delivers the transaction context (Coordinator)
 */
public Coordinator getContext()

/**
 * calls current location's goTo
 * to migration to a new location,
 * <B>can be used by agent programmers!</B>
 *
 * @param dest_name name of the destination
 */
protected final void migrateTo(LocationName dest_name)

private class StandardAgentThread
    extends MoleThread

/**
 * a worker calls this method.
 * Calls deactivateAgent to stop all threads of the agent except
 * for the active thread.
 */
protected removeAgentThread()
```

---

## Class Location

```
/**
 * This class implements MOLE locations
 * an agent to mark it.
 */
public class Location
    extends UnicastRemoteObject
    implements RemoteLocation

// Messages
private int messages = 1000;           // the initial size of Vector mails

private Vector mails = new Vector(messages); // the mailbox of
                                           // this location

private Hashtable messagesrep = new Hashtable(); // the dictionary of
                                                  // the mess. rep.

/**
 * send a message asynchronously
 */
void message(Message m)

/**
 * sends a message with errorsemantics 0
 *
 * @param receiver the name of the receiver of the message
 * @param receiverLocation the name of the receiver's location
 * @param mid the message id
 * @param o the content of the message
 */
void sendUnreliable(AgentName receiver, LocationName
                    receiverLocation, long mid, Serializable o)

/**
 * sends a message with errorsemantics 1
 *
 * @param receiver the name of the receiver of the message
 * @param receiverLocation the name of the receiver's location
 * @param mid the message id
 * @param o the content of the message
 */
void sendReliable(AgentName receiver, LocationName receiverLocation,
                  long mid, Serializable o)
```

---

```
/**
 * sends a message with errorsemantics 2
 *
 * @param receiver the name of the receiver of the message
 * @param receiverLocation the name of the receiver's location
 * @param mid the message id
 * @param o the content of the message
 */
void sendMailbox(AgentName receiver, LocationName receiverLocation,
                 long mid, Serializable o)

/**
 * delivers a message to a receiver
 *
 * @param m the message to receive
 */
public void deliverMessage(Message m)
    throws RemoteException

/**
 * sends a message synchronously
 */
public void syncmessage(Message m)

/**
 * returns the number of messages for this agent
 */
public int numberOfMessages(AgentName name)

/**
 * deliver the messages for this agent
 */
public void deliverMessagesFor(Agent anAgent)

/**
 * sets the messages representative of the client<BR>
 * client = AgentName of Agent who wishes to be represented<BR>
 * representative = name of an agent which gets the messages instead of client<BR>
 * when client isn't here<BR>
 * when representative == null : removes representative entry
 */
public void setMessagesRepresentative(AgentName client,
                                       AgentName representative)
```

---

```
// processes messages to the location itself
private void automessage(Message m)

// store a message until the receiver is present
// or the expiration date is over
private void storeMessage(Message m)

/**
 * dipatches an incoming method call
 *
 * @param rpcm information for moleview
 * @param procName the name of the called method
 * @param remoteAgent a badge or agent name that identifies the called
 * agent
 * @param param an array of parameters for the called method
 */
public Serializable dispatchRPC(RPCMessage rpcm, String procName,
                                PeerQualifier remoteAgent,
                                Serializable[] param)

/**
 * calls a remote method with no parameters
 *
 * @param caller the name of the calling agent
 * @param procName the name of the remote procedure
 * @param remoteAgent an agent that provides the called method
 * @param remoteLocation the destination location
 * @return the serializable object returned by the remote method
 */
public Serializable call(AgentName caller, String procName,
                          PeerQualifier remoteAgent, LocationName remoteLocation)
    throws NoSuchReceiverException, NoSuchLocationException,
    NoSuchMethodException, RemoteException, UnknownHostException,
    NotBoundException, IllegalAccessException, InvocationTargetException

/**
 * calls a remote method with one parameter
 *
 * @param caller the name of the calling agent
 * @param procName the name of the remote procedure
 * @param remoteAgent an agent that provides the called method
 * @param remoteLocation the destination location
 * @param p1 the parameter for the remote method
 * @return the serializable object returned by the remote method
 */
```

```
public Serializable call(AgentName caller, String procName,
                        PeerQualifier remoteAgent,
                        LocationName remoteLocation, Serializable p1)
    throws NoSuchReceiverException, NoSuchLocationException,
           NoSuchMethodException, RemoteException, UnknownHostException,
           NotBoundException, IllegalAccessException, InvocationTargetException
```

```
/**
```

```
 * calls a remote method with two parameters
 *
 * @param caller the name of the calling agent
 * @param procName the name of the remote procedure
 * @param remoteAgent an agent that provides the called method
 * @param remoteLocation the destination location
 * @param p1,p2 the parameters for the remote method
 * @return the serializable object returned by the remote method
 */
```

```
public Serializable call(AgentName caller, String procName,
                        PeerQualifier remoteAgent,
                        LocationName remoteLocation, Serializable p1,
                        Serializable p2)
    throws NoSuchReceiverException, NoSuchLocationException,
           NoSuchMethodException, RemoteException, UnknownHostException,
           NotBoundException, IllegalAccessException, InvocationTargetException
```

```
/**
```

```
 * calls a remote method with three parameters
 *
 * @param caller the name of the calling agent
 * @param procName the name of the remote procedure
 * @param remoteAgent an agent that provides the called method
 * @param remoteLocation the destination location
 * @param p1,p2,p3 the parameters for the remote method
 * @return the serializable object returned by the remote method
 */
```

```
public Serializable call(AgentName caller, String procName,
                        PeerQualifier remoteAgent,
                        LocationName remoteLocation, Serializable p1,
                        Serializable p2, Serializable p3)
    throws NoSuchReceiverException, NoSuchLocationException,
           NoSuchMethodException, RemoteException, UnknownHostException,
           NotBoundException, IllegalAccessException, InvocationTargetException
```

```
/**
 * calls a remote method with four parameters
 *
 * @param caller the name of the calling agent
 * @param procName the name of the remote procedure
 * @param remoteAgent an agent that provides the called method
 * @param remoteLocation the destination location
 * @param p1,p2,p3,p4 the parameters for the remote method
 * @return the serializable object returned by the remote method
 */
public Serializable call(AgentName caller, String procName,
                          PeerQualifier remoteAgent,
                          LocationName remoteLocation, Serializable p1,
                          Serializable p2, Serializable p3, Serializable p4)
    throws NoSuchReceiverException, NoSuchLocationException,
    NoSuchMethodException, RemoteException, UnknownHostException,
    NotBoundException, IllegalAccessException, InvocationTargetException

/**
 * calls a remote method with five parameters
 *
 * @param caller the name of the calling agent
 * @param procName the name of the remote procedure
 * @param remoteAgent an agent that provides the called method
 * @param remoteLocation the destination location
 * @param p1,p2,p3,p4,p5 the parameters for the remote method
 * @return the serializable object returned by the remote method
 */
public Serializable call(AgentName caller, String procName,
                          PeerQualifier remoteAgent,
                          LocationName remoteLocation, Serializable p1,
                          Serializable p2, Serializable p3, Serializable p4,
                          Serializable p5)
    throws NoSuchReceiverException, NoSuchLocationException,
    NoSuchMethodException, RemoteException, UnknownHostException,
    NotBoundException, IllegalAccessException, InvocationTargetException

/**
 * calls a remote method with an unspecified number of parameters
 *
 * @param caller the name of the calling agent
 * @param procName the name of the remote procedure
 * @param remoteAgent an agent that provides the called method
 * @param remoteLocation the destination location
 * @param param an array with the parameters for the remote method
 * @exception NoSuchReceiverException, if the remote agent isn't found
```

---

```

* @exception NoSuchLocationException, if the remote location isn't found
* @exception NoSuchMethodException, if no suitable method is found
* @exception RemoteException, if the registry couldn't be contacted
* @exception UnknownHostException, if the destination host isn't found
* @exception NotBoundException, if the destination location's dispatchRPC
* method isn't bound to the registry
* @exception IllegalAccessException, if there were problems accessing
* items on the remote engine
* @exception InvocationTargetException, if the called method has thrown
* an exception during execution
* @return the serializable object returned by the remote method
*/
public Serializable call(AgentName caller, String procName,
                        PeerQualifier remoteAgent,
                        LocationName remoteLocation, Serializable[] param)
    throws NoSuchReceiverException, NoSuchLocationException,
    NoSuchMethodException, RemoteException, UnknownHostException,
    NotBoundException, IllegalAccessException, InvocationTargetException

/**
 * sends a message to another location
 *
 * @param m the message to send
 */
private void lowlevelsend(Message m)

/**
 * only used by arrive() and createNewAgent(Hashtable params)
 *
 * @param anAgent the Agent
 * @param async true if asynchronous
 * @return true if started, false if agent is null
 */
private boolean _arrive(Agent anAgent, boolean async)

/**
 * sets the RPC representative of the client<BR>
 * client = AgentName of agent who wishes to be represented<BR>
 * representative = name of an agent which gets the rpc instead of client
 * when client isn't here<BR>
 * when representative == null : removes representative entry
 */
public void setRPCRepresentative(AgentName client, AgentName representative)

```

```
/*
 * Orphan Communication
 */
/**
 * this method is used by the orphan control. If the location of the receiver
 * is identical to the sender location, the method OrphanControlNotify will
 * be called directly. If the sender and receiver location are at the same
 * engine, the method OrphanCall of the receiver location will be called.
 * Otherwise, a rpc call to the receiver location will be made.
 * @return true if the call successful, false otherwise
 */
public Serializable OrphanCall( RPCMessage rpcm )
    throws NoSuchReceiverException, NoSuchLocationException,
        NoSuchMethodException, RemoteException, UnknownHostException,
        NotBoundException, IllegalAccessException, InvocationTargetException

/**
 * an agent calls this procedure when he wants to migrate
 * (default migration method)
 */
boolean goTo(Agent traveller, LocationName place)

/**
 * called remotely to handle an incoming migration
 *
 * @param arrAgent the migrating agent
 * @return true, if the migration was successful
 * <dd> null, otherwise
 */
public boolean handleMigration(Agent arrAgent)

/**
 * executes the agent.
 *
 * @param agent the agent
 */
public boolean executeAgent(Agent agent)

/**
 * stops all threads of the agent except for the active thread
 *
 * @param traveller the agent
 */
void deactivateAgent(Agent traveller)
```

```
/**
 * returns a Reference to his MoleviewAgent, <BR>
 * create a new Agent if it not already exists
 * @return Reference to MoleviewAgent
 */
public synchronized MoleviewAgent getMoleviewAgent()

/**
 * start agents asynchronously
 */
protected void startAgents()
```

---

## Class Session

```
/**
 * This class implements sessions.
 */
public class Session
    implements Serializable

/**
 * sends a session message to the remote peer of this session
 *
 * @param content the content of the message
 * @exception SessionNotActiveException, if this session is not active
 * anymore
 */
public void message(Serializable content)
    throws SessionNotActiveException

/**
 * calls the method with the name procName of the remote peer.
 *
 * @param procName the name of the method to call
 * @see call(String, Serializable[])
 */
public Serializable call(String procName)
    throws SessionNotActiveException, NoSuchReceiverException,
        NoSuchLocationException, NoSuchMethodException, RemoteException,
        UnknownHostException, NotBoundException, IllegalAccessException,
        InvocationTargetException
```

---

```
/**
 * calls the method with the name procName and the parameter p1
 * of the remote peer.
 *
 * @param procName the name of the method to call
 * @param p1 the parameter to the called method
 * @see call(String, Serializable[])
 */
public Serializable call(String procName, Serializable p1)
    throws SessionNotActiveException, NoSuchReceiverException,
    NoSuchLocationException, NoSuchMethodException, RemoteException,
    UnknownHostException, NotBoundException, IllegalAccessException,
    InvocationTargetException

/**
 * calls the method with the name procName and the arguments p1, p2
 * of the remote peer.
 *
 * @param procName the name of the method to call
 * @param p1,p2 the parameters to the called method
 * @see call(String, Serializable[])
 */
public Serializable call(String procName, Serializable p1, Serializable p2)
    throws SessionNotActiveException, NoSuchReceiverException,
    NoSuchLocationException, NoSuchMethodException, RemoteException,
    UnknownHostException, NotBoundException, IllegalAccessException,
    InvocationTargetException

/**
 * calls the method with the name procName and the arguments p1, p2, p3
 * of the remote peer.
 *
 * @param procName the name of the method to call
 * @param p1,p2,p3 the parameters to the called method
 * @see call(String, Serializable[])
 */
public Serializable call(String procName, Serializable p1, Serializable p2, Serializable p3)
    throws SessionNotActiveException, NoSuchReceiverException,
    NoSuchLocationException, NoSuchMethodException, RemoteException,
    UnknownHostException, NotBoundException, IllegalAccessException,
    InvocationTargetException

/**
 * calls the method with the name procName and the arguments p1, p2, p3,
 * p4 of the remote peer.
 *
 * 
```

```
* @param procName the name of the method to call
* @param p1,p2,p3,p4 the parameters to the called method
* @see call(String, Serializable[])
*/
public Serializable call(String procName, Serializable p1, Serializable p2, Serializable p3,
    Serializable p4)
    throws SessionNotActiveException, NoSuchReceiverException,
    NoSuchLocationException, NoSuchMethodException, RemoteException,
    UnknownHostException, NotBoundException, IllegalAccessException,
    InvocationTargetException

/**
 * calls the method with the name procName and the arguments p1, p2, p3,
 * p4, p5 of the remote peer.
 *
 * @param procName the name of the method to call
 * @param p1,p2,p3,p4,p5 the parameters to the called method
 * @see call(String, Serializable[])
 */
public Serializable call(String procName, Serializable p1, Serializable p2, Serializable p3,
    Serializable p4, Serializable p5)
    throws SessionNotActiveException, NoSuchReceiverException,
    NoSuchLocationException, NoSuchMethodException, RemoteException,
    UnknownHostException, NotBoundException, IllegalAccessException,
    InvocationTargetException

/**
 * calls the method with the name procName and the arguments param
 * of the remote peer.
 *
 * @param procName the name of the method to call
 * @param param an array of parameters to the called method
 * @exception SessionNotActiveException, if this session is not active
 * anymore
 * @exception NoSuchReceiverException, if the remote agent isn't found
 * @exception NoSuchLocationException, if the remote location isn't found
 * @exception NoSuchMethodException, if no suitable method is found
 * @exception RemoteException, if the registry couldn't be contacted
 * @exception UnknownHostException, if the destination host isn't found
 * @exception NotBoundException, if the destination location's dispatchRPC
 * method isn't bound to the registry
 * @exception IllegalAccessException, if there were problems accessing
 * items on the remote engine
 * @exception InvocationTargetException, if the called method has thrown
 * an exception during execution
 *
 */
```

```
*/
public Serializable call(String procName, Serializable[] param)
    throws SessionNotActiveException, NoSuchReceiverException,
    NoSuchLocationException, NoSuchMethodException, RemoteException,
    UnknownHostException, NotBoundException, IllegalAccessException,
    InvocationTargetException
```

---

### Class **ErrorMessage**

```
// this class represents an error message.
// ErrorMessage has an error semantics of 0
public class ErrorMessage
    extends Message
```

---

### Class **MulticastMessage**

```
/**
 * This class implements multicast messages, which are used to
 * send messages to all providers of a specific badge
 */
public class MulticastMessage
    extends Message
```

---

### Class **MessageThread**

```
/**
 * MessageThread.java
 *
 * duty: subclass of MoleThread to represent syncmessages
 */
public class MessageThread
    extends MoleThread
```

---

### Class **Mail**

```
/**
 * a message plus a mstime for Location
```

```
*/  
public class Mail  
    extends Object
```

---

### Class **UnformattedMessage**

```
// this class implements a message which consists of a string and which  
// is handled more efficient than normal messages  
public class UnformattedMessage  
    extends Message
```

---

### Class **Message**

```
// this class implements the message objects in Mole  
public class Message  
    extends Object  
    implements Serializable  
  
/**  
 * the error semantics of the message  
 */  
// 0 = do nothing if this message can't delivered  
// 1 = send an error message (with errorsemantics 0) back to the  
//     sender if an error occurred  
// 2 = store this message in the location's mailbox, if the receiver isn't  
//     available  
public int errorsemantics = 0;  
  
/**  
 * the content of the message  
 */  
public Serializable content = null;  
  
/**  
 * constructs an empty message  
 */  
public Message()
```

```
// the standard constructor of this class
/**
 * constructs a message
 *
 * @param s the sender's name
 * @param sl the name of the sender's location
 * @param r the receiver
 * @param rl the name of the destination location
 * @param es the errorsemantics of this message
 * @param mid the message id
 * @param o the content of this message
 */
public Message(AgentName s, LocationName sl, PeerQualifier r, LocationName rl, int es,
                long mid, Serializable o)
// the standard constructor of this class without automatic created message id

/**
 * constructs a message without a message id
 *
 * @param s the sender's name
 * @param sl the name of the sender's location
 * @param r the receiver
 * @param rl the name of the destination location
 * @param es the errorsemantics of this message
 * @param o the content of this message
 */
public Message(AgentName s, LocationName sl, PeerQualifier r, LocationName rl, int
                es, Serializable o)

/**
 * converts this message to an ErrorMessage.
 *
 * @param s the sender of the ErrorMessage
 * @param sl the name of the sender's location
 * @param r the reason for the error
 */
public ErrorMessage toErrorMessage(AgentName s, LocationName sl, String r)
```

---

## Class MoleprotWrapper

```
/**
 * abstract class providing access to some internal variables of a MOLE
 * engine, so that MoleprotServer may report it to a monitor
 */
public abstract class MoleprotWrapper
    extends mole.util.moleview.Moleprot

/**
 * get error condition of message
 */
public String getMessageCondition( Message m )

/**
 * get contents of message
 */
public String getMessageContents( Message m )
```

---

## Interface **RemoteLocation**

```
/**
 * This is the remote interface for locations.
 */
public interface RemoteLocation
    extends Remote

/**
 * delivers a message at the receiving location.
 *
 * @param m the message to deliver
 * @exception RemoteException If the location could not be contacted.
 */
void deliverMessage(Message m) throws RemoteException;
```

---

## Class **IllegalRPCException**

```
// An attempt has been made to call a procedure of a User Agent
public class IllegalRPCException
    extends RemoteException

// Constructs an IllegalRPCException with no detail message
IllegalRPCException()
```

```
// Constructs an IllegalRPCException with the specified detail message
IllegalRPCException(String s)
```

---

### **Class TransactionContextException**

```
// An attempt has been made to use a call-method with the first
// parameter of the called method not being a Coordinator
public class TransactionContextException
    extends Exception
```

```
// Constructs an TransactionContextException with no detail message
TransactionContextException()
```

```
// Constructs an TransactionContextException with the specified detail message
TransactionContextException(String s)
```

---

### **Class MoleThread**

```
/**
 * create a new thread
 */
public static void threadCreate(Runnable target)
```

---

### **Class IllegalThreadCreateException**

```
// An attempt has been made from a mobile Agent to create a thread
public class IllegalThreadCreateException
    extends Exception
```

```
// Constructs an IllegalThreadCreateException with no detail message
IllegalThreadCreateException()
```

```
// Constructs an IllegalThreadCreateException with the specified detail message
IllegalThreadCreateException(String s)
```

---

### **Interface Periodical**

---

```
/**
 * Agents which want to have a heartbeat must implement this interface
 * @see Agent
 * @see HeartbeatAgentThread
 */
public interface Periodical
```

---

### Class MCP

```
/**
 * A Master Control Program (-) which schedules threads in a time-slice
 * manner. Mole need this construct because JDK 1.0 (Solaris) don't have
 * a system thread implementation
 */

public class MCP
    extends Thread

/**
 * handle threads in threadgroup according to given handle_mode
 *
 * @param threadgroup the threadgroup to be handled
 * @param handle_mode the handle mode
 */
private synchronized void handleAgentThreadGroup( AgentThreadGroup threads,
                                                    int handle_mode )
```

## Anhang "A.2"

Im Anhang "A.2" befinden sich eine Ausführliche Beschreibung der im Kapitel 5 aufgeführten Klassen, Methoden, vom Agentensystem Mole 3.0, sowie auch Variablen und Methoden die zusätzlich hinzugefügt wurden.

### Class Agent

```

/**
 * The father of all agents. The root in of the class hierarchy from which
 * all agents must inherit.
 */
public abstract class Agent
    extends Object
    implements Serializable

// the number of locations for the next stage
protected int numberNextStage;

// default value for the number of locations for the next stage
protected int defaultNumberNextStage;

// contains the current stage
protected Vector currentStage = new Vector();

/**
 * <B>can be used by agent programmers!</B>
 * Sets the number of locations for the next stage.
 *
 * @param number  number of the locations for the next stage.
 * @return int    the number of the next stage.
 */
protected int numberOfNextStage(int number)

/**
 * constructor for an agent class.
 * In the constructor the global number of locations for the next stage is determined from
 * Location.
 */
protected Agent()

/**
 * constructor for an agent class.
 * In the constructor the global number of locations for the next stage is determined from
 * Location.

```

```
* @param description decorator of the agent needed for orphan detection
*/
protected Agent(String description)

// This method is called if, after the next stage was constructed with a one-time value set by
// numberOfNextStage(), the next but one stage should be constructed with the default
// value again.
protected void resetNumberNextStage()
```

---

## Class Location

```
/**
 * This class implements MOLE locations
 * an agent to mark it.
 */
public class Location
    extends UnicastRemoteObject
    implements RemoteLocation

// default number for the next stage of all agents
static private int globalDefaultNumberNextStage = 3;

// contains all B_Entries whose preconditions are true
protected Vector result;

// contains all B_Entries that represent the next stage
protected Vector endTotal = new Vector();

// get-method to read out the value
static protected int getGlobalDefaultNumberNextStage()

/**
 * only used by arrive() and createNewAgent(Hashtable params)
 * @param anAgent the Agent
 * @param async true if asynchronous
 * @return true if started, false if agent is null
 */
private boolean _arrive(Agent anAgent, boolean async)

private void startAgent(Agent anAgent)
```

---

```

/**
 * is used to determine the next stage
 * @param anAgent the agent.
 */
public void stageConstruction(Agent anAgent)

/**
 * checks if there are B_Entries in the endTotal-vector
 * with the same location name.
 * If true it deletes the B_Entry with the lower priority.
 */
private void B_EntryLocationName()

/**
 * checks if the locaton name is contained in vector.
 * @param vector the vector to check
 * @param locName the location name
 * @return boolean true if the location name is not in the vector
 * false otherwise
 */
boolean notInVector(Vector vector, LocationName locName)

/**
 * delivers the index of the first occurence of null in the vector.
 * The search begins with the specified index. If no null is detected -1 is returned.
 * @param vector the vector to search.
 * @param index the index of the beginning of the search.
 * @return int the index of the first occurrence of null. -1 if no null is detected.
 */
private int indexOfNull(Vector vector, int index)

```

---

## Class Engine

```

/**
 * The top object of a Java Mole runtime environment,
 * an engine may have a set of locations.
 */
public class Engine
    extends Object

// contains the location names read from the configuration
// file "knownloc.cfg".
static Vector knownLoc = new Vector();

```

```
// reads the location names from "knownloc.cfg"
private void readConfig()

private void init(String cfgfile, boolean no_cli)

/**
 * startNewAgent takes a String containing an agent class, a hash table
 * containing parameters and a string containing the location on which to
 * start the agent. The hash table has to contain the tags as keys and the
 * parameters for the agent init() method as entries.
 *
 * @param agentClass the class of the agent as a String.
 * @param parameters the parameters for the init()-method in a hashtable.
 * @param location the location on which to start the agent.
 * @return true if the agent has been started
 *         false if either the format was not correct or an
 *         error occurred starting the agent.
 */
public static boolean startNewAgent(String agentClass,
                                     Hashtable parameters, String location)
```

## Anhang "B"

### Projektplan für die Diplomarbeit

Titel: Fehlertoleranz in Mole  
Autor: Papoulidis Konstantinos  
Datum: 19.09.1999

#### 1 Einleitung

Der Projektplan dient dem Bearbeiter und dem Betreuer als Grundlage für die Projektüberwachung und -steuerung. Er legt den Projektablauf aber nicht ein für allemal fest, sondern soll im Laufe der Arbeit ergänzt und angepaßt werden. So ist es z.B. möglich, daß anfänglich spezifizierte Arbeitspakete weiter verfeinert oder Arbeitspakete zugunsten anderer aufgegeben werden.

Der Projektplan kann in Absprache mit dem Betreuer in beiderseitigem Einverständnis in den folgenden Fällen angepaßt werden:

- a) bei Verzug,
- b) bei vorzeitigem Abschluß eines Arbeitspakets oder
- c) bei Gewinn neuer Erkenntnisse.

Der aktualisierte Projektplan soll in den Anhang der Ausarbeitung einfließen.

**Projektbeschreibung:** Bei diesem Projekt handelt es sich um eine Diplomarbeit in der Abteilung Verteilte Systeme am Institut für Parallele und Verteilte Höchstleistungsrechner der Universität Stuttgart.

Unter Mobilien Agenten versteht man "Programmkonstrukte", die die Fähigkeit haben, autonom zu handeln und von Rechner zu Rechner zu migrieren, falls auf beiden Seiten ein "Agentensystem" installiert ist. Vorbedingung zum Einsatz der Agententechnologie in kommerziellem Umfeld ist das Vorhandensein von Mechanismen zur Steigerung der Fehlertoleranz, welche die Ausführung des Agenten auch unter Vorhandensein von Fehlern, sowohl von Netzwerk als auch von ausführenden Rechnern, sicherstellen. In dieser Arbeit soll das in der Abteilung verteilte Systeme entwickelte Agentensystem Mole um solche Mechanismen erweitert werden.

Grundlage dieser Arbeit ist die vorliegende Implementation eines fehlertoleranten Protokolles zur Sicherstellung der Exactly-Once-Ausführung von Agenten. Dieses soll in dieser Arbeit in

Mole integriert werden. Da das Protokoll das Ausführungsmodell eines Agenten wesentlich einschränkt, sind hierfür weitgehende Änderungen in der Mole-API durchzuführen.

Grundlegende Idee des fehlertoleranten Mechanismus ist, daß die Ausführung eines Agenten auf einem Rechner (Arbeiter) durch mehrere andere Rechner (Beobachter) überwacht wird. Die vorliegende Implementation des Protokolls erwartet, daß der Arbeiter und die Beobachter bei jeder Migration des Agenten explizit angegeben wird. Die in einer parallelen Arbeit in Mole integrierte Erweiterung, welche eine sehr flexible Möglichkeit der Agenten - Reiserouten - Spezifikation zur Verfügung stellt, erlaubt eine weitestgehend automatische Bestimmung des Arbeiters und der Beobachter für die nächste Migration. Für diese automatische Bestimmung ist ein Algorithmus zu entwerfen (weitestgehend schon vorgegeben), dessen Anforderungen an die Reiserouten-Spezifikation (als Vorgabe für deren parallel laufende Implementation) zu spezifizieren und den Algorithmus zu implementieren.

Die Implementierung erfolgt in Java und C++. Das gegebene Protokoll verwendet Transaktionen, hierfür wird der CORBA Transaction Service verwendet. Bei der Implementierung sind die Projektrichtlinien der Abteilung zu beachten. Die Ergebnisse der Arbeit sind in einem Abschlußvortrag zu präsentieren.

Laufzeit: 15.4.99 - 14.10.99

Betreuer: Dipl.-Inform. Markus Straßer

Prüfer: Prof. Dr. rer. nat. Kurt Rothermel

In den nachfolgenden Abschnitten werden zunächst die Arbeitspakete identifiziert und beschrieben, dann wird ein Zeitplan zu deren Durchführung festgelegt und schließlich erfolgt die Definition der Meilensteine und der dafür zu erstellenden Dokumente.

## **2 Beschreibung der Arbeitspakete**

Dieser Abschnitt umfaßt die Definition der wesentlichen Arbeitspakete, die Abhängigkeiten zwischen einzelnen Paketen und die Identifikation von kritischen Punkten, die die Durchführung des Projektes gefährden können.

### **2.1 Definition der Arbeitspakete**

Die im Rahmen der Diplomarbeit durchzuführenden Tätigkeiten lassen sich in folgende, zusammenhängende Arbeitspakete aufgliedern:

AP1, Einarbeitung: Genaue Einarbeitung in die Aufgabenstellung (Motivation, Ziel).  
Status: abgeschlossen.

AP2, Projektplan: Erstellung eines Projektplanes.  
Status: abgeschlossen.

AP3 Schwerpunkt "Mole-API": Änderungen in der Mole API durchführen zur sinnvollen Integration des Exactly-Once-Protokolls, Entwurf (in der Objektmodellierungssprache UML = Unified Modeling Language), Codierung.  
Status: abgeschlossen.

AP4, Schwerpunkt "weiterleiten": Algorithmus zur Bestimmung des Workers und der Beobachter der nächsten Stufe **entwerfen**, Anforderungen an die Reiserouten-Spezifikation spezifizieren.  
Status: abgeschlossen.

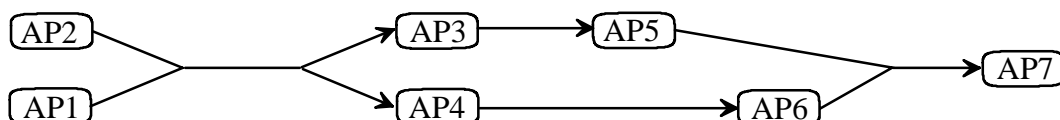
AP5, Schwerpunkt "einbauen": Exactly-Once-Protokoll einbauen in MOLE 3.0 Entwurf (UML), Codierung.  
Status: in Arbeit.

AP6, Schwerpunkt "weiterleiten": Algorithmus zur Bestimmung des Workers und der Beobachter der nächsten Stufe **implementieren**.  
Status: abgeschlossen.

AP7, Ausarbeitung: Erstellung der Ausarbeitung.  
Status: in Arbeit.

## 2.2 Abhängigkeiten der Arbeitspakete

Zwischen den Arbeitspaketen der Diplomarbeit bestehen die folgenden Abhängigkeiten:



Aus logischer Sicht werden die Arbeitspakete AP3, AP4, AP5, AP6 nach Vollendung der Arbeitspakete AP1, AP2 durchgeführt sowie AP7 nach allen anderen Arbeitspaketen. Die Arbeitspakete AP1 und AP2 laufen zeitlich Parallel ab. Zur sinnvollen Integration des Protokolls ist AP5 nach AP3 zu realisieren. Im AP4 wird der Entwurf des Algorithmus sowie die Anforderungen an die Reiseroutenspezifikation zeitlich parallel zum AP3 abgewickelt. Die Codierung im AP6 ist nach dem Entwurf im AP4 abzuwickeln.

## 2.3 Kritische Punkte / Risiken

Keine.

## 3 Zeitplan

In diesem Abschnitt erfolgt die zeitliche Planung der Arbeitspakete und Meilensteine. Der Bearbeitungszeitraum ist 6 Monate oder 26 Wochen. Wenn sich der Zeitraum mehrerer Arbeitspakete überlappt, dann werden diese parallel bearbeitet. Am Ende des Bearbeitungszeitraums ist ein Puffer eingeplant worden, der für Verzögerungen oder andere unvorhergesehene Tätigkeiten genutzt werden soll.

geplanter Zeitraum	Aufwand	Arbeitspakete und Meilensteine	tatsächlicher Zeitraum
15.4.- 26.5.	6 PW	AP1 Einarbeitung, AP2 Projektplan <b>Meilenstein 1</b>	15.4. - 2.6.
27.5.- 30.6.	5 PW	AP3 Schwerpunkt "Mole-API", (Entwurf, Codierung) AP4 Schwerpunkt "weiterleiten", (Entwurf und Spezifikation der Anforderungen an die Reiserouten-Spezifikation) <b>Meilenstein 2</b>	3.6. - 11.8.
1.7.- 4.8.	5 PW	AP5 Schwerpunkt "einbauen" (Entwurf, Codierung) <b>Meilenstein 3</b>	17.8. - 13.10.
5.8.- 1.9.	4 PW	AP6 Schwerpunkt "weiterleiten", (Codierung) <b>Meilenstein 4</b>	12.8. - 19.9.
2.9.- 6.10.	5 PW	AP7 Ausarbeitung <b>Meilenstein 5</b>	2.9. - 13.10.
7.10.- 13.10.	1 PW	Puffer	

(PW = Personenwoche)

## 4 Meilensteine und Dokumente

Als Ergebnis der jeweiligen Arbeitspakete sollen zu den Meilensteinen folgende Dokumente entstehen, die nach diesen einer Versionskontrolle unterliegen und nur in Absprache mit dem Betreuer geändert werden sollen.

Zu den Meilensteinen sind die folgenden Dokumente zu erstellen:

26.5.1999 , Meilenstein 1: Projektplan, Literaturliste bezogen auf Einarbeitung.

30.6.1999 , Meilenstein 2: Entwurf (UML) und Quellcode zu AP3 Schwerpunkt "Mole API", Entwurf (UML) zu AP4 Schwerpunkt "weiterleiten" als auch Spezifikationsdokument bezüglich den Anforderungen an die Reiserouten-Spezifikation.

4.8.1999, Meilenstein 3: Entwurf (UML) und Quellcode zu AP5 Schwerpunkt "einbauen".

1.9.1999, Meilenstein 4: Quellcode zu AP6 Schwerpunkt "weiterleiten".

6.10.1999, Meilenstein 5: Ausarbeitungsdokument.

## 5 Fazit

In diesem Projektplan wurden die durchzuführenden Arbeiten identifiziert, der jeweilige Zeitaufwand abgeschätzt, Meilensteine und Termine definiert. Anhand dieses Projektplans konnte festgestellt werden, dass das Projekt in Verzug geraten war und die Anfangs definierten Termine nicht eingehalten werden konnten. Die Arbeitspakete 3 und 4 erforderten zum Beispiel einen höheren Aufwand als anfangs dafür vorgesehen war. Das Problem konnte zwar frühzeitig erkannt werden, eine geeignete Gegenmaßnahme aber konnte nicht ergriffen werden, da die beiden Arbeitspakete, wichtige Bestandteile des Projekts waren, auf denen die Arbeitspakete 5 und 6 aufbauten. Das Arbeitspaket 3 diente der sinnvollen Integration des Protokolls in Mole und konnte so nicht zugunsten eines anderen Arbeitspakets aufgegeben oder nur zum Teil bearbeitet werden. Das Arbeitspaket 4 enthielt den Entwurf, um die Durchführung des Arbeitspakets 6 zu ermöglichen. Somit war ich gezwungen das Arbeitspaket 5 parallel zum Arbeitspaket 6 sowie darüber hinaus zu bearbeiten.

### **Meine persönlichen Erkenntnisse aus diesem Projekt sind:**

Eine Planung am Anfang eines Projekts ist sehr schwierig, wenn eigene Erfahrungswerte mit Projekten fehlen. Natürlich ist ein Projektplan sehr hilfreich, um bei weiteren Projekten die gewonnenen Erkenntnisse einzusetzen, eventuelle methodische Fehler zu vermeiden und sobald neue Informationen über das Projekt vorliegen, den Projektplan anzupassen. Um ein Softwareprojekt überschaubar zu halten und möglichst effektiv durchführen zu können, sollte dieses vorher sorgfältig geplant werden.

## Anhang "C": Literaturverzeichnis

- [Baumann et al. 97] Baumann J., Hohl F., Rothermel K., Straßer M.:  
**Mole - Concepts of a Mobile Agent System.**  
"Mobility - Processes, Computers and Agents",  
acm Press, Addison-Wesley, 1999, S. 535-554.
- [Baumann et al. 98] Baumann J., Hohl F., Rothermel K., Schwehm M., Straßer M.:  
**Mole 3.0: A Middleware for Java-Based Mobile Software Agents.**  
In Proceedings Middleware'98, Springer Verlag, London, 1998.
- [Buschle 99] Buschle Jürgen:  
**Reiserouten-Konzepte für Mobile Agenten.**  
Studienarbeit Nr. 1754, Universität Stuttgart, Institut für Parallele und  
Verteilte Höchstleistungsrechner, Lehrstuhl für Verteilte Systeme, 1999
- [Flanagan 97] Flanagan David:  
**Java in a Nutshell.**  
Verlag O' Reilly & Associates, 2. Auflage, Köln, 1997.
- [FowKen 98] Fowler Martin, Kendall Scott:  
**UML konzentriert.**  
Addison-Wesley-Longman, Bonn, 1998.
- [Friedel 98] Friedel Klaus:  
**Fehlertolerantes Protokoll zur Exactly-Once-Ausführung  
von Agenten.**  
Diplomarbeit Nr. 1651, Universität Stuttgart, Institut für Parallele und  
Verteilte Höchstleistungsrechner, Lehrstuhl für Verteilte Systeme, 1998
- [Hohl 95] Hohl Fritz:  
**Konzeption eines einfachen Agentensystems und Implementation  
eines Prototyps.**  
Diplomarbeit Nr. 1267, Universität Stuttgart, Institut für Parallele und  
Verteilte Höchstleistungsrechner, Lehrstuhl für Verteilte Systeme, 1995
- [HolSchu 97] Holz Antje, Schumann Michael:  
**Das Programmierhandbuch Java 1.1.**  
Sybex - Verlag, 2. Auflage, Düsseldorf, 1997.
- [IONA 98a] IONA Technologies PLC:  
**OrbixWeb Programmer's Guide.**  
IONA Technologies PLC, 1998.

- [IONA 98b] IONA Technologies PLC:  
**OrbixOTM Guide.**  
IONA Technologies PLC, 1998.
- [IONA 98c] IONA Technologies PLC:  
**OrbixOTS Programmer's and Administrators's Guide.**  
IONA Technologies PLC, 1998.
- [Javasoft] Homepage von Javasoft  
<http://www.javasoft.com>
- [Kubach 97] Kubach Uwe:  
**Redesign/Reimplementation der Kommunikationsmechanismen in Mole.**  
Studienarbeit Nr. 1640, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Lehrstuhl für Verteilte Systeme, 1997
- [LemPer 97] Lemay Laura, Perkins Charles L.:  
**Java 1.1 in 21 Tagen.**  
Markt & Technik - Verlag, München, 1997.
- [Maihöfer 97] Maihöfer Christian:  
**Ein Protokoll zur Wahrung der Exactly-Once-Eigenschaft Mobiler Agenten.**  
Diplomarbeit Nr. 1565, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, Lehrstuhl für Verteilte Systeme, 1997
- [Melchisedech 96] Melchisedech Ralf, Folien zur Vorlesung:  
**Konzeption und Aufbau objektorientierter Software.**  
WS 1996, Universität Stuttgart, Institut für Informatik, Abteilung Software Engineering.
- [Mole] Homepage von "Mole"  
<http://www.informatik.uni-stuttgart.de/ipvt/vs/projekte/mole.html>
- [Reißing 97] Reißing Ralf:  
Folien zum **Java Kompaktkurs.**  
14.07.1997 - 18.07.1997, Universität Stuttgart, Institut für Informatik, Abteilung Software Engineering.
- [OMG 98] CORBA services: Common Object Services Specification.  
<http://www.omg.org>, December 1998.
- [Rosenberger 98] Rosenberger Jeremy:  
**CORBA in 14 Tagen.**  
Markt & Technik - Verlag, München, 1998.

- [RotStr 98] Rothermel K., Straßer Markus:  
**A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents.**  
In Proceedings 17th IEEE Symposium on Reliable Distributed Systems, 1998, (SRDS'98), IEEE Computer Society, Los Alamitos, California, S. 100 - 108.
- [Rumbaugh et al. 93] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorenzen W.:  
**Objektorientiertes Modellieren und Entwerfen.**  
Carl Hanser - Verlag und Prentice-Hall International,  
Wien - London, 1993.
- [Straßer 98] Straßer Markus:  
**Reliability Concepts for Mobile Agents.**  
International Journal of Cooperative Information Systems (IJCIS),  
Volume 7, Number 4, World Scientific, 1998, S. 355-382.
- [SUN] Homepage von Sun - Microsystems  
<http://www.sun.com/>
- [Willms 98a] Willms Andre:  
**C-Programmierung lernen.**  
Addison-Wesley-Longman, Bonn, 1998.
- [Willms 98b] Willms Andre:  
**C++ -Programmierung.**  
Addison-Wesley-Longman, Bonn, 1998.

