Universität Stuttgart
Fakultät Informatik

Prüfer:	Prof. K. Rothermel
Betreuer:	Dipl.-Inform. M. Straßer
Beginn:	1. Januar 1998
Ende:	30. Juni 1998
CR-Klassifikation	C.2.4, D.4.4, D.4.5

Diplomarbeit Nr. 1624

Konzeption und Implementation eines zuverlässigen und skalierbaren Agentenservers

Michael Bader

Institut für Parallele und Verteilte Höchstleistungsrechner

Breitwiesenstraße 20-22

D-70565 Stuttgart

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung

Kapitel 1

1.1 Motivation und Aufgabenstellung

Mobile Agenten sind aktive Objekte, die sich selbständig durch Rechnernetze bewegen und ihre Ausführung auf anderen Rechnern fortsetzen können. Grundlage für die Fortbewegung und Ausführung der Agenten ist aber das Vorhandensein eines Agentensystems auf den zu nutzenden Rechnerknoten. Dieses stellt den Agenten die nötige Infrastruktur zur Verfügung, um u. a. auf andere Rechnerknoten migrieren oder mit anderen Agenten kommunizieren zu können.

Das Konzept der mobilen Agenten ist ein relativ neuer, vielversprechender Ansatz im Bereich der verteilten Systeme. Um die Agententechnologie auch im kommerziellen Umfeld einsetzen zu können, bedarf es einiger Vorbedingungen. Neben den vielfältigen Sicherheitsaspekten, um sowohl Agenten, wie auch ausführende Rechner zu schützen, müssen auch Zuverlässigkeit und Skalierbarkeit der Orte, an denen Agenten ausgeführt werden, gewährleistet werden.

Für die Realisierung eines solchen zuverlässigen und skalierbaren Agentensystems bietet sich die Himalaya-Architektur von Tandem an. Neben der Zuverlässigkeit, die u.a. auf Grund von Hard- und Softwareredundanz erreicht wird, bietet sie mit dem Konzept der Serverklasse ein mächtiges Mittel zur Realisierung skalierbarer Systeme. Eine Serverklasse ist eine Menge von Prozessen, die alle dasselbe Serverprogramm ausführen. Ein Transaktionsmonitor verteilt Kundenanfragen auf die Prozesse der Serverklasse.

Ziel dieser Arbeit war es, ein zuverlässiges und skalierbares Agentensystem auf Basis eines am IPVR verfügbaren Tandem-Himalaya-Rechners unter Verwendung des Transaktionsmonitors Tuxedo zu entwerfen und zu implementieren. Dieses System sollte in Bezug auf die agentenspezifischen Funktionen (Kommunikation und Migration) konzeptionell möglichst eng an das an der Universität Stuttgart entwickelte mobile Agentensystem MOLE angelehnt sein. Sich aus den Eigenheiten der Entwicklungsplattform ergebende notwendige Abweichungen sollten jedoch berücksichtigt werden. Die Realisierung sollte mit der Programmiersprache Java erfolgen.

1.2 Gliederung dieser Arbeit

Diese Arbeit ist folgendermaßen gegliedert:

In Kapitel zwei werden die Grundzüge des an der Universität Stuttgart entwickelten mobilen Agentensystems MOLE erläutert. Dabei werden auch dessen Nachteile vorgestellt, die bei der Entwicklung des neuen Agentensystems beseitigt werden sollten.

Kapitel drei befaßt sich mit der Entwicklungsplattform, auf der das neue System zu entwickeln war. Es werden Architektur des verwendeten Rechners, dessen Betriebssystem, der Transaktionsmonitor Tuxedo und die Java-Umgebung beschrieben.

In Kapitel vier wird dann die Konzeption des neuen Agentensystems vorgestellt. Neben der neuen Architektur und den sich daraus ergebenden Problemen, werden auch die angebotenen Kommunikations- und Mobilitätskonzepte ausführlich dargestellt.

Kapitel fünf beschreibt die Implementation des Agentensystems. Hier werden einige interessante Aspekte bei der Realisierung der in Kapitel vier vorgestellten Konzepte detaillierter geschildert.

Das sechste Kapitel führt eine Bewertung des neuen Agentensystems durch.

Zum Abschluß stellt Kapitel sieben einige Ideen vor, die bei künftigen Arbeiten am neuen Agentensystem aufgegriffen werden könnten.

Das Agentensystem MOLE

Kapitel 2

Die Konzeption des im Rahmen dieser Diplomarbeit zu entwickelnden mobilen Agentensystems sollte nicht bei Null anfangen. Vielmehr waren die Erfahrungen bei der Arbeit an und mit dem an der Universität Stuttgart entwickelten Agentensystem MOLE zu berücksichtigen. Deshalb wird dieses System nachfolgend kurz vorgestellt und auf dessen Mängel näher eingegangen. Für eine detailliertere Darstellung der Grundkonzepte von MOLE sei auf [BaHoSt] und [BHRS] verwiesen.

2.1 Das Verarbeitungsmodell von MOLE

MOLE ist ein in der Programmiersprache Java realisiertes Agentensystem. Entitäten des Systems sind, wie bei vielen anderen Agentensysteme auch, Orte (Locations) und Agenten. Orte sind abstrakte Plätze, auf denen sich Agenten aufhalten können. Sie stellen Agenten neben den grundlegenden Diensten des Agentensystems, nämlich Kommunikation und Migration, weitere ortsspezifische Dienste zur Verfügung. Außerdem können auf ihnen Agenten erzeugt werden. Dies kann durch Benutzereingaben, aber auch programmgesteuert durch andere Agenten erfolgen. Ein Ort kann nicht über verschiedene Rechnerknoten verteilt werden. Es ist aber möglich mehrere Orte auf einem Rechner anzusiedeln.

Agenten sind spezielle Java-Programme. In MOLE gibt es zwei Arten von Agenten: Useragenten und Systemagenten. Erstgenannte sind mobil, d.h. sie können sich durch Migration an andere Orte begeben. Die Entscheidung zur Migration wird dabei vom Agenten selbst und nicht durch das System, z.B. aus Gründen der Lastverteilung, getroffen. Agenten haben die Möglichkeit lokal, d.h. mit Agenten des eigenen Aufenthaltsortes, aber auch global, d.h. mit sich auf anderen Plätzen aufhaltenden Agenten, zu kommunizieren. Letzteres wird dabei als “teurer” als die lokale Kommunikation betrachtet. Neben der Migration und der Kommunikation können Agenten neue Agenten erzeugen und die vom Ort angebotenen Dienste nachfragen. Da Systemagenten ortsspezifische Dienste anbieten, sind sie immobil. Sie stellen den Useragenten die Möglichkeit zur Verfügung, durch agenten-typische Kommunikation auf diese Dienste zuzugreifen. Da Systemagenten zur Erfüllung ihrer Aufgaben in der Regel

einen weiterreichenden Zugriff auf Systemressourcen benötigen, sind sie im Vergleich zu Useragenten mit mehr Rechten ausgestattet und können nur vom Administrator des Ortes gestartet werden. Sowohl User- als auch Systemagenten können mehrere Ausführungsstränge (Threads) besitzen.

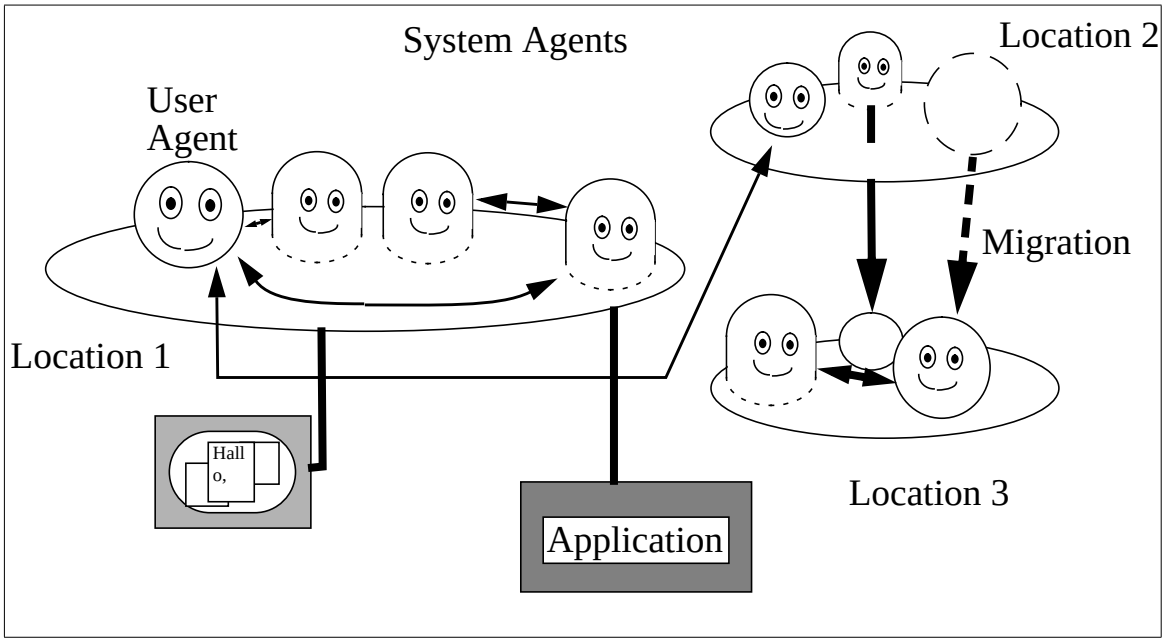


Abbildung 2-1: Verarbeitungsmodell von MOLE [BaHoSt]

Abb. 2-1 ist [BaHoSt] entnommen und veranschaulicht noch einmal das MOLE zu Grunde liegende Modell aus Orten, User- und Systemagenten.

Als Kommunikationsmechanismen stehen in MOLE der Nachrichtenversand und der entfernte Prozeduraufruf (RPC, RMI) zur Verfügung. Agenten können Nachrichten synchron oder asynchron an andere Agenten verschicken, aber auch öffentliche Methoden anderer Agenten aufrufen. Die Zielagenten können sich dabei auch auf anderen Orten befinden. Der Kommunikationspartner kann entweder durch den global eindeutigen Agentennamen oder durch sogenannte Badges spezifiziert werden. Badges sind Marken (einfache Zeichenketten), die sich Agenten anheften können, um anderen Agenten vorhandene Eigenschaften, wie z.B. angebotene Methoden bzw. Dienste oder eine Gruppenzugehörigkeit, zu signalisieren. Bei der Adressierung durch eine Badge wird aus der Menge der Agenten, die mit dieser Badge gekennzeichnet sind, der Kommunikationspartner ausgewählt. Zusätzlich zu Agentennamen bzw. Badge muß immer auch der Name des Aufenthaltsortes des Zielagenten angegeben werden, da dieser zur Zeit nicht aus dem Agentennamen abgeleitet werden kann.

Idealerweise sollte ein Agent nach einer erfolgreichen Migration an der Stelle weiterarbeiten, die dem Aufruf zur Migration folgt. Dazu muß aber neben dem Programmcode und den Daten auch der Ausführungszustand des Agenten an

den Zielort übertragen werden. Die Ermittlung und das Wiedereinspielen des Ausführungszustandes würde Änderungen am Java-Interpreter erfordern und ist nicht ohne größeren Programmieraufwand möglich. Außerdem wäre das Agentensystem dann nicht mehr auf unveränderten Java-Interpretern lauffähig. Der veränderte Interpreter müßte mit dem Agentensystem ausgeliefert und installiert werden. MOLE bietet deshalb nur die Möglichkeit einer “schwachen” Migration an. Hierbei werden nur Programmcode und Daten des Agenten übertragen, nicht aber dessen Ausführungszustand. Die Ausführung des Agenten am Zielort wird durch Aufruf einer festgelegten Methode begonnen.

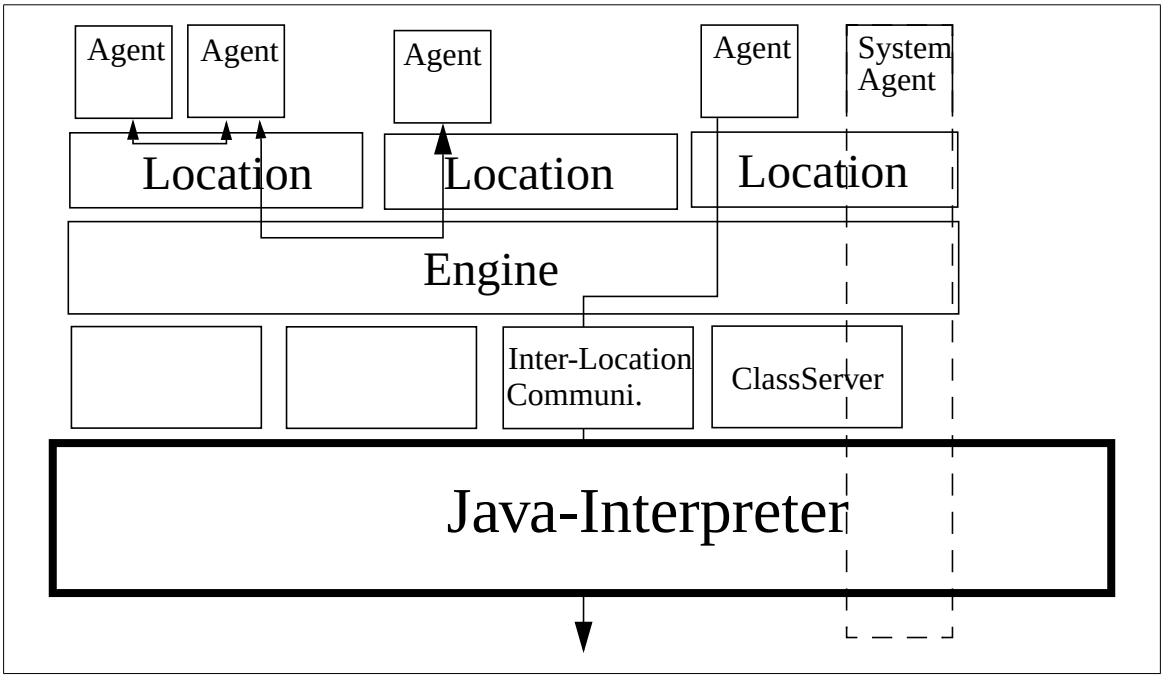


Abbildung 2-2: Architektur von MOLE [BaHoSt]

Die aus [BaHoSt] stammende Abb. 2-2 zeigt die Architektur von MOLE. Während ein Ort (Location) Agenten verwaltet und ausführt, ist eine Engine für die Verwaltung eines oder mehrerer Orte zuständig. Sie stellt diesen gemeinsame Dienste, wie z.B. einen Class-Server oder die Kommunikation zwischen Orten, zur Verfügung. Der Class-Server ist dabei für die Beschaffung von Java-Klassen, die bei der Ausführung von Agenten benötigt werden, zuständig. Das Aufsetzen von Orten auf eine Engine ist für Agenten transparent. Eine Engine wird durch einen Java-Interpreter (Java-Virtual-Machine, JVM) ausgeführt. Dabei laufen die Agenten aller Orte einer Engine als Threads in einem einzigen Prozeß ab.

2.2 Nachteile von MOLE

Das hier beschriebene Modell weist bei näherem Hinsehen einige nicht zu unterschätzende Mängel auf, die im folgenden beschrieben werden.

2.2.1 Zuverlässigkeit

So gibt es im gesamten Lebenszyklus eines Agenten keinen Zeitpunkt, zu dem dieser in einem persistenten Zustand vorliegt. Einzige Ausnahme ist das Herunterfahren eines Ortes. In diesem Fall werden alle sich auf ihm befindenden Agenten in einer Datei gespeichert. Beim späteren Hochfahren des Ortes wird diese Datei gelesen und die darin enthaltenen Agenten wieder zur Ausführung gebracht. Da die Agenten nur gespeichert werden, wenn der Ort und damit sie selbst außer Betrieb genommen werden, soll dieser Fall nicht weiter berücksichtigt werden. Stürzt der Prozeß, in dem der Agent ausgeführt wird, durch Software- oder Hardwareausfälle ab, so geht der Agent mitsamt seinen Daten verloren. Im Hinblick auf den Einsatz von Agentensystemen im kommerziellen Umfeld, bei dem Agenten dann z.B. auch elektronisches Geld mit sich führen können, ist ein solcher Verlust untragbar, da er unter anderem zu einem unüberwindbaren Akzeptanzproblem führen würde.

Durch die Autonomie der Agenten gerade bei Migrationen ist es in einem weit vernetzten und unzuverlässigen Agentensystem nicht ohne weiteres möglich festzustellen, ob ein Agent abgestürzt oder noch mit seinen Berechnungen beschäftigt ist. Dieses Problem kann durch periodische Zustandsmeldungen des Agenten an seinen Benutzer bzw. Programmierer gelöst werden, verursacht aber einen zusätzlichen Aufwand sowohl bei der Agentenprogrammierung, als auch bei der Ressourcennutzung. Würde das Agentensystem die zuverlässige Ausführung des Agenten sicherstellen, so könnte dieser Aufwand eingespart werden.

2.2.2 Skalierbarkeit

Wie oben beschrieben, werden sämtliche Agenten aller Orte einer Engine in einem einzigen Prozeß ausgeführt. Dies hat zunächst den Vorteil, daß aufwendige Prozeßwechsel entfallen können, und die Kommunikation dieser Agenten untereinander durch lokale Methodenaufrufe innerhalb derselben JVM, und damit relativ effizient, vonstatten gehen kann. Als Nachteil steht dem gegenüber, daß ein einzelner fehlerhafter Thread alle übrigen Threads des Prozesses beeinträchtigen und in Mitleidenschaft ziehen kann. Schwerer wiegt aber die Einschränkung der Skalierbarkeit des Agentensystems, also der Fähigkeit auf gewachsene Anforderungen an das System, beispielsweise durch eine höhere Anzahl der zu verwaltenden Agenten, angemessen reagieren und diese befriedigen zu können. Da die Agententhreads einer Engine an nur einen Prozeß und damit an den diesen ausführenden Prozessor gebunden sind, können sie nicht parallel ausgeführt werden und vorhandene zusätzliche Prozessoren in einem Mehrprozessorsystem bleiben ungenutzt. Bei Rechnersystemen, welche die Ausführung der Threads eines Prozesses durch verschiedene Prozessoren unterstützen, trifft die vorangegangene Aussage zwar nicht zu. Aber auch bei

diesen Systemen kann die Ausführung der Agenten einer Engine auf Grund der Bindung an einen Prozeß nicht auf mehrere vernetzte Rechner verteilt und somit auch keine Lastverteilung durchgeführt werden.

Neben der Zuverlässigkeit spielt auch die Skalierbarkeit beim kommerziellen Einsatz von Agentensystemen eine wichtige Rolle. So ist die Akzeptanz von kommerziellen Diensten vor allem auf Grund von Sicherheitsaspekten zur Zeit noch gering. Mit dem Einsatz geeigneter Verfahren wird aber für die Zukunft mit einem deutlichen Akzeptanzgewinn und einem damit einhergehenden starken Anstieg der Nachfrage dieser Dienste gerechnet. Die diesen Diensten zugrunde liegende Software sollte daher in der Lage sein, die sich daraus ergebenden größeren Anforderungen zu bewältigen.

Die eben aufgeführten Nachteile von MOLE bildeten die Motivation für die Entwicklung eines neuen Agentensystems, das diese Mängel beseitigt. Bevor dessen Konzeption vorgestellt wird, werden aber zuerst die Eigenschaften und Möglichkeiten der Entwicklungsplattform erläutert.

Die Entwicklungsplattform

Kapitel 3

In diesem Kapitel wird die Tandem-Himalaya-Plattform, auf der das Agentensystem zu entwickeln war, beschrieben. Die sich aus dieser Rechnerarchitektur und der darauf aufsetzenden Entwicklungsumgebung ergebenden Möglichkeiten werden später im Kapitel “Konzeption des Agentensystems” dazu verwendet, die im vorangegangenen Kapitel erläuterten Nachteile von MOLE zu beseitigen. Im einzelnen werden Rechnerarchitektur, Betriebssystem und darauf aufsetzende Module, der Transaktionsmonitor Tuxedo und die Java-Anbindung vorgestellt.

3.1 Rechnerarchitektur des Tandem-Himalaya-K10000-Servers

Begonnen wird mit der Beschreibung des am IPVR verfügbaren Tandem Himalaya K10000 Rechners, auf dem das Agentensystem realisiert werden sollte. Die in diesem und den beiden folgenden Abschnitte aufgeführten Informationen basieren dabei auf den Angaben in [Tand95a]und [Tand95b]. Der K10000 Rechner ist ein lose gekoppeltes Mehrprozessorsystem, das mit zwei bis 16 Prozessoren bestückbar ist, die durch Nachrichtenaustausch miteinander kommunizieren. Abb. 3-1 zeigt die Architektur dieses Systems, dessen einzelne Komponenten nachfolgend beschrieben werden.

Im K10000 Server werden NSR-N (NonStop System RISC Model N) Prozessoren, die auf RISC (Reduced Instruction Set Computer) basieren, eingesetzt. Jeder einzelne davon enthält alle benötigten Bestandteile eines Computersystems und ist daher in der Lage auch unabhängig von den anderen Prozessoren zu arbeiten. Ein Prozessor besteht aus:

- Einer Verarbeitungseinheit (Instruction Processing Unit, IPU), die Befehle und Daten aus dem Hauptspeicher holt und die Daten entsprechend den Befehlen verarbeitet.
- Dem Hauptspeicher (Memory), in dem Programmbefehle und Daten gespeichert werden. Die IPU kann Ein-/Ausgabegeräten, wie z.B. Festplatten, Bandlaufwerke oder Terminals, nicht direkt ansprechen, deshalb müssen Daten von bzw. an diese Geräte im Hauptspeicher abgelegt werden.
- Einem oder mehreren Ein-/Ausgabekanälen (I/O Channel), die für den Transfer von Daten zwischen Ein-/Ausgabegeräten und dem Hauptspeicher verantwortlich sind.
- Der Schnittstelle zum Interprozessorbus (IPB), welcher der Kommunikation zwischen den Prozessoren dient.
- Einer unabhängigen Stromversorgung (Power Supply) und Batterie.

Das Hauptziel beim Entwurf dieser Rechnerarchitektur war von Anfang an die Schaffung eines möglichst fehlertoleranten Systems. Deshalb kommen neben den Prozessoren, die auch unabhängig voneinander arbeiten können, zwei voneinander unabhängige Interprozessorbusse, dual-ported Gerätekontroller und gespiegelte dual-ported Festplatten zum Einsatz. Auf Grund dieser durchgängigen Redundanz der Hardwarekomponenten kann der Ausfall einer einzelnen Komponente durch das Systems aufgefangen werden. Abb. 3-1 kann beim Veranschaulichen dieses Sachverhalts helfen:

- Beim Versagen des einen Interprozessorbusses kann der Verkehr über den anderen geleitet werden.
- Beim Ausfall eines Prozessors kann ein anderer dessen Aufgaben übernehmen (dazu später mehr).
- Beim Ausfall eines Ein-/Ausgabekanals kann ein anderer die betroffenen Gerätekontroller verwalten.
- Beim Ausfall eines Gerätekontrollers können die betroffenen Geräte von einem anderen gesteuert werden.
- Beim Ausfall einer gespiegelten Festplatte führt der Gerätekontroller Schreib- und Leseoperationen auf der verbliebenen Platte aus.

Auch das Risiko, daß das Rechnersystem durch einen Stromausfall komplett zum Stehen kommt, kann durch das Versorgen der Prozessoren durch voneinander unabhängige Stromquellen abgemildert werden. Ein nicht zu unter-

schätzender Nebeneffekt ist dabei die Möglichkeit, einzelne Hardwarekomponenten z.B. zu Wartungs- oder Reparaturzwecken zu entfernen, ohne das ganze System herunterfahren zu müssen. Zusätzlich zur Stromversorgung verfügen die Prozessoren über eine Batterie, die im Notfall den Hauptspeicher je nach Kapazität bis zu einigen Stunden mit Strom versorgen kann.

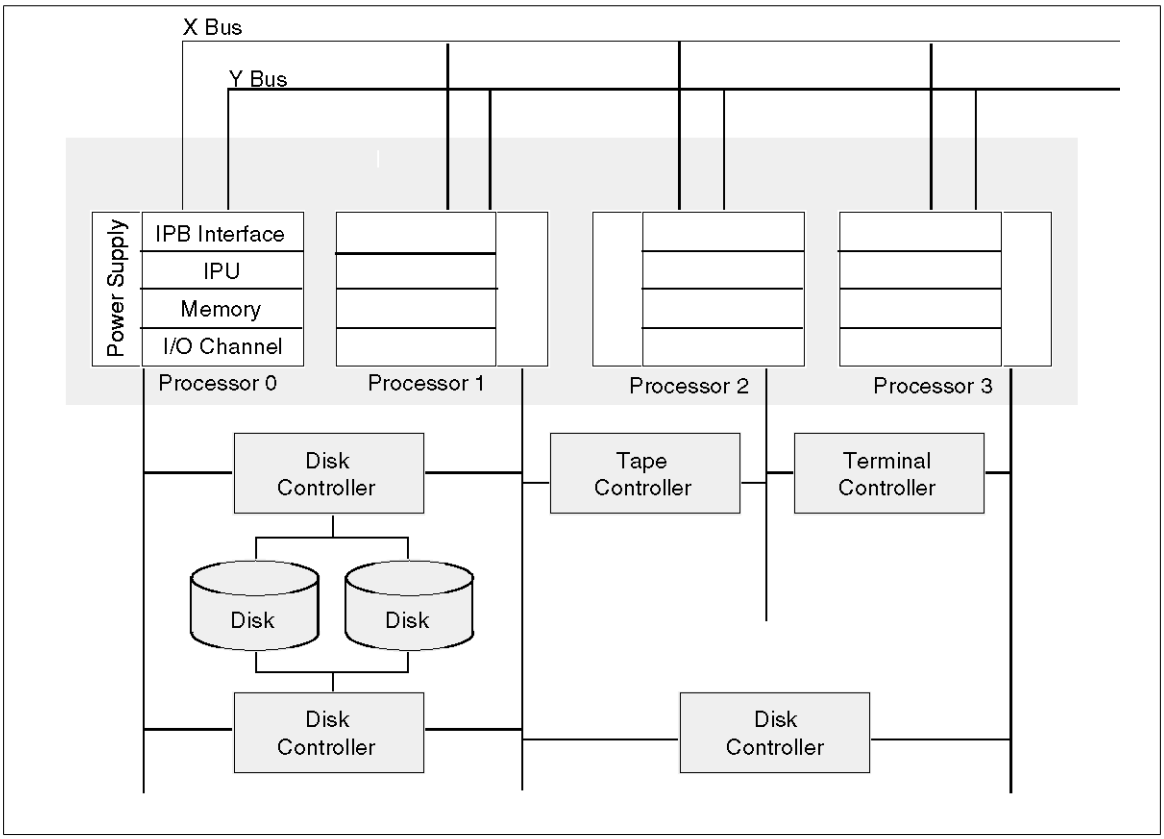


Abbildung 3-1: Architektur des Tandem-Rechners [Tand95a]

Jeder Prozessor führt umfangreiche Selbsttests durch. Einige der dabei feststellbaren Fehler können vom Prozessor selbständig behoben werden, wie beispielsweise Speicherfehler durch ECC (Error Correcting Codes). Andere werden dem Betriebssystem gemeldet, das dann in adäquater Weise darauf reagieren kann. Wie im nächsten Abschnitt beschrieben, besitzt das Betriebssystem einen weiteren Mechanismus um fehlerhafte Prozessoren zu erkennen.

Neben der Fehlertoleranz wurde auch an die Skalierbarkeit des Systems gedacht. Wird eine höhere Leistung benötigt so kann der K10000-Rechner in sinnvollen Schritten mit weiteren Prozessoren aufgerüstet werden. Im Gegensatz zu fest gekoppelten Systemen, bei denen auf Grund des Overheads, der bei der Kommunikation der Prozessoren über den gemeinsamen Hauptspeicher entsteht, ab etwa sieben bis acht Prozessoren das Hinzufügen eines Prozessors zu keiner nennenswerten Leistungssteigerung mehr führt, hat das Aufrüsten

mit Prozessoren bei einem lose gekoppelten System deutliche Leistungssteigerungen zur Folge. Reicht die Rechenkraft eines Servers mit seinen maximal 16 Prozessoren noch immer nicht aus, so kann er durch das Expand-Subsystem mit weiteren Tandem-NonStop-Systemen vernetzt werden. Auch der Aufbau eines auf Grund seiner Topologie TorusNet genannten massiv parallelen Systems mit bis zu gut 4000 Prozessoren ist möglich.

3.1.1 Betriebssystem

Wie die Beschreibung der Rechnerarchitektur bereits erahnen läßt, handelt es sich beim Betriebssystem des Himalaya-K10000-Servers nicht um eine monolithische Einheit, sondern um ein verteiltes Betriebssystem. Es besteht aus einer Menge von Prozessen, im folgenden zur Unterscheidung von Benutzerprozessen Systemprozesse genannt. Diese Systemprozesse kooperieren untereinander durch Nachrichtenaustausch über ein gemeinsames Nachrichtensystem. Die grundlegenden Systemprozesse, wie z.B. Prozeßmonitor oder Speicherverwaltung, laufen unabhängig voneinander in jedem einzelnen Prozessor ab. Andere Systemprozesse hingegen werden nur in einigen Prozessoren ausgeführt. Dazu gehören beispielsweise Ein-/Ausgabeprozesse zur Ansteuerung von Peripheriegeräten, die nur in den über ihre Ein-/Ausgabekanäle mit den Geräten verbundenen Prozessoren ablaufen.

Die Kommunikation zwischen den Prozessen erfolgt, wie bereits erwähnt, nachrichtenbasiert und zwar über den Interprozessorbus, der alle im System vorhandenen Prozessoren miteinander verbindet. Adressiert werden die Prozesse nicht hardwarenah, sondern über symbolische Namen. Das Betriebssystem führt eine Tabelle, die diese Namen auf die konkreten Positionen, auf denen die Prozesse ansprechbar sind, abbildet. Dies hat den Vorteil, daß z.B. zu Zwecken der Lastverteilung oder nach Ausfall eines Prozessors, Prozesse ohne Probleme auf andere Prozessoren verschoben werden können.

Bereits bei der Konzeption des NonStop-Kernels wurde die Vernetzung von mehreren Systemen berücksichtigt und ist daher in Form des Expand-Subsystems integraler Bestandteil des Betriebssystems. Auf Grund dieses Sachverhalts und dem Aufbau eines Rechners aus mehreren voneinander unabhängigen Prozessoren, der eher an ein Rechnernetz denken läßt, stellt die Vernetzung mehrerer Tandem-NonStop-Systeme nicht eine völlig neue Sichtweise, sondern eine natürliche Erweiterung des Einzelsystems dar. Das Adressieren von Prozessen, die nicht auf dem lokalen System ablaufen, geschieht ebenfalls über die oben erwähnte Tabelle, die für diese Prozesse zusätzlich den zugehörigen Rechnernamen enthält. Der Hauptunterschied zwischen Einzelsystem und vernetztem System besteht darin, daß Prozesse innerhalb des glei-

chen Einzelsystems über den Interprozessorbus Nachrichten austauschen, während der Nachrichtenaustausch zwischen Prozessen verschiedener Systeme über externe Verbindungen abgewickelt wird.

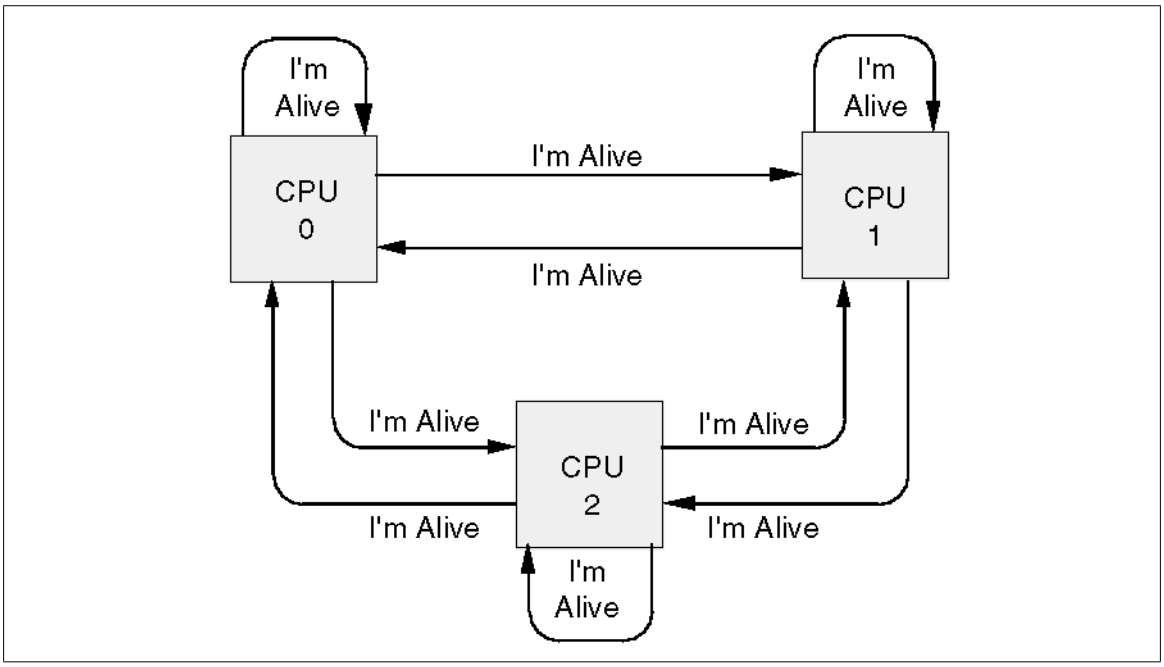


Abbildung 3-2: "I'm alive"-Nachrichten [Tand95a]

Um die Fehlertoleranz, die das Himalaya System bietet, auszunutzen, müssen fehlerhafte Komponenten erst einmal entdeckt werden. Im letzten Abschnitt wurden die von den Prozessoren durchgeführten Selbsttests erwähnt. Eine weitere Möglichkeit, die auf dem Nachrichtensystem beruht, stellt das Betriebssystem zur Verfügung und ist in Abb. 3-2 dargestellt. Der Teil des Betriebssystems, der in allen Prozessoren abläuft, geht dafür nach dem folgenden Algorithmus vor:

1. Jeder Prozessor sendet periodisch "I'm alive"-Nachrichten an sich selbst und alle anderen Prozessoren im System. Diese Nachrichten sollen den Prozessoren die korrekte Funktion des Senders signalisieren.
2. Innerhalb des gleichen periodischen Intervalls prüft jeder Prozessor den Empfang der "I'm alive"-Nachrichten aller Prozessoren des Systems.
3. Bemerkt ein Prozessor das Ausbleiben dieser Nachrichten eines anderen Prozessors für die Zeitdauer zweier solcher Intervalle, so kontaktiert er die anderen Prozessoren, und der fehlerhafte Prozessor wird außer Betrieb genommen
4. Die anderen Prozessoren übernehmen, wie weiter unten beschrieben, die Prozesse des fehlerhaften.

Mit der Erkennung eines Fehlers ist es nicht getan. Auf ihn muß möglichst bald reagiert werden, um Schäden zu vermeiden. Das Betriebssystem bietet in diesem Zusammenhang die Möglichkeit Prozesse als Prozeßpaare auszuführen. Der Primärprozeß eines Prozeßpaars ist aktiv und führt den Programmcode aus. Der Backup-Prozeß des Prozeßpaars belegt zwar Speicher in einem anderen Prozessor, ist aber im Normalzustand passiv. Vor jeder kritischen Operation, wie z.B. einem Festplattenzugriff, sendet der Primär- dem Backup-Prozeß sogenannte Checkpoint-Nachrichten, um den Backup-Prozeß auf den aktuellen Stand der Programmausführung zu bringen. Fällt der Prozessor, der den Primärprozeß ausführt, aus, so nimmt der auf einem anderen Prozessor residierende Backup-Prozeß die Arbeit ab dem letzten gültigen Checkpoint wieder auf. Er arbeitet dabei mit den Daten, die auch der Primärprozeß benutzte. Abb. 3-3 stellt das Konzept der Prozeßpaare noch einmal dar. Die meisten Systemprozesse sind als Prozeßpaare realisiert. Aber auch Benutzerprozesse können auf diese Weise ausgeführt werden.

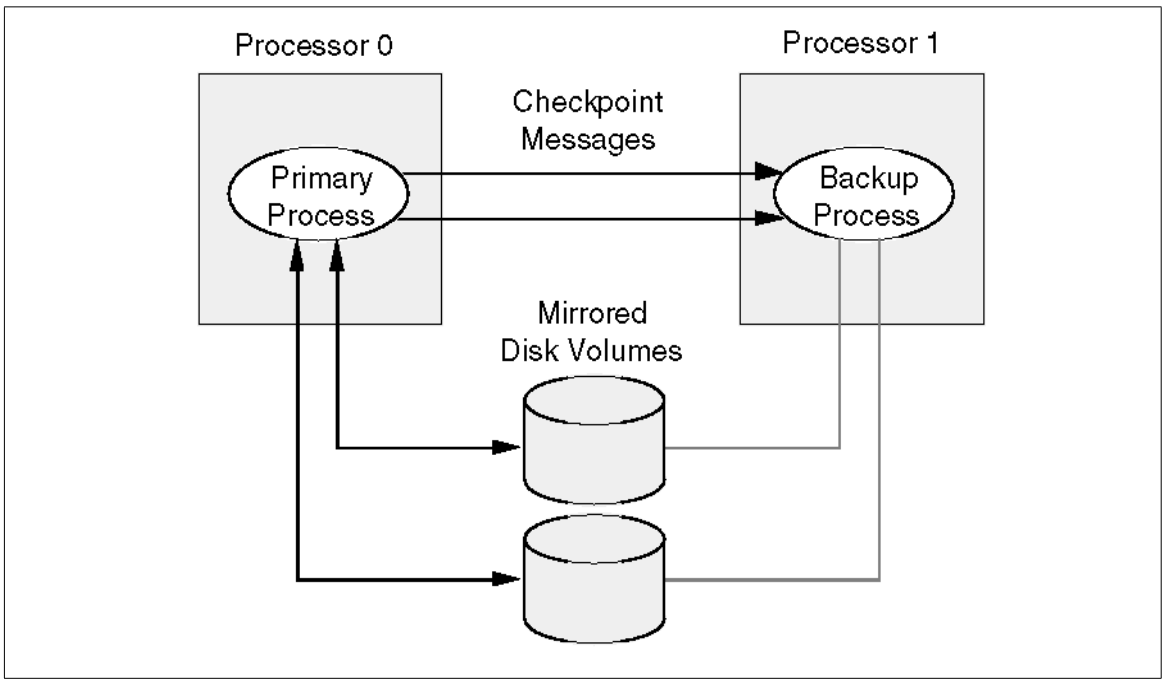


Abbildung 3-3: Konzept der Prozeßpaare [Tand95a]

Tandem stellt dem Benutzer zwei Umgebungen für den Zugang zum Betriebssystem zur Verfügung. Die Guardian-Umgebung ist die ursprüngliche und proprietäre. Seit neuerem gibt es aber mit den Open System Services (OSS) eine weitere Umgebung, die wie Guardian den Zugriff auf Dienste des Betriebssystems sowohl über eine Kommandozeile, als auch über ein API (Application Programming Interface) ermöglicht. Auf Grund der Ähnlichkeit von OSS mit UNIX haben Applikationen, die auf OSS basieren, dabei den Vorteil der relativ

leichten Portierbarkeit auf andere UNIX-Umgebungen. Außerdem fällt die Einarbeitung in OSS Personen, die bereits Erfahrungen mit UNIX gesammelt haben, leichter als die in Guardian.

3.1.2 Weitere Komponenten der Tandem Entwicklungsumgebung

Auf das Betriebssystem setzen weitere Softwarekomponenten auf. Mit NonStop SQL/MP steht ein relationales Datenbank Managementsystem (DBMS) zur Verfügung, das auf SQL (Structured Query Language) basiert. Das Nonstop TM/MP Paket enthält einen Transaktionsmanager, der für das Deklarieren von Transaktionen, deren Verwaltung und die Einhaltung der Transaktionseigenschaften verantwortlich ist. Außerdem ist Nonstop TM/MP für die Verwaltung von Sperren und Recovery-Maßnahmen im Falle von Systemabstürzen zuständig.

Da im Laufe dieser Arbeit wiederholt von Transaktionen die Rede ist, soll der Begriff der Transaktion hier definiert werden. Eine Transaktion ist eine ununterbrechbare Folge von Anweisungen bzw. Operationen für die folgende Eigenschaften (ACID-Eigenschaften) gelten:

- Atomizität (Atomicitiy): Eine Transaktion wird entweder komplett oder gar nicht durchgeführt (“alles oder nichts”).
- Konsistenz (Consistency): Befinden sich die Ressourcen (z.B. eine Datenbank) in einem konsistenten Zustand, so befinden sich die Ressourcen auch nach Abschluß der Transaktion wieder in einem konsistenten Zustand.
- Isolation: Eine Transaktion wird durch nebenläufige Transaktionen nicht beeinträchtigt. Die Auswirkungen der Transaktionen entsprechen einer seriellen Ausführungsreihenfolge der Transaktionen.
- Dauerhaftigkeit (Durabiity): Die Änderungen einer abgeschlossenen Transaktion bleiben dauerhaft erhalten.

Ein weiteres Softwarepaket, NonStop TS/MP, bietet einige grundlegende Dienste an, die bei der Ausführung von OLTP-Anwendungen (OnLine Transaction Processing) benötigt werden. Dazu gehört das Management von Serverprozessen und die Anbindung von Client- an Serverprozesse. So werden bei höherer Last weitere Serverprozesse erzeugt, abgestürzte automatisch neu gestartet und nicht mehr benötigte beendet. Außerdem stellt NonStop TS/MP die Kommunikationspfade zwischen Client- und Serverprozessen her und verteilt die Anfragen der Clients an die Serverprozesse, um Durchsatz und Antwortzeit zu optimieren..

3.2 NonStop-Tuxedo

Tuxedo ist ein Softwaresystem zur Entwicklung, Ausführung und Administration von verteilten Applikationen auf der Basis des Client/Server-Modells. Tuxedo wurde ursprünglich an den Bell Laboratories von AT&T entwickelt, wird aber seit 1996 von BEA vermarktet. Seit der Markteinführung im Jahre 1983 wurde es kontinuierlich weiterentwickelt, und viele technische Innovationen hielten in die neueren Versionen Einzug. Heute ist es das führende System im Online-Transaction-Processing-Bereich. Tuxedo besteht aus mehreren Komponenten. Die wichtigste stellt der Transaktionsmonitor System/T dar, auf den gleich näher eingegangen wird. Weitere Komponenten sind mit System/D eine SQL-Datenbank und mit System/Q Message-Queues.

Mit NonStop-Tuxedo steht ein speziell an die Tandem-NonStop-Rechnersysteme angepaßtes Tuxedo zur Verfügung. Es setzt auf die im letzten Abschnitt beschriebenen Komponenten und das Betriebssystem auf und kann dadurch die damit verbundenen Eigenschaften, wie parallele Verarbeitung, gute Skalierbarkeit und hohe Verfügbarkeit, nutzen. Abb. 3-4 zeigt dieses Zusammenspiel (PTP, Pathway und Enscribe sind alternative Transaktionsmonitore bzw. eine Datenbank, auf die hier nicht weiter eingegangen wird). So werden z.B. NonStop-Tuxedo-Server als NonStop TS/MP Serverklassen realisiert. Eine Serverklasse ist dabei eine Menge von Prozessen, die alle dasselbe Serverprogramm ausführen. Eine Serverklasse hat den Vorteil, daß diese Serverprozesse in mehreren Prozessoren gestartet werden können und somit eine Lastverteilung erreicht wird.

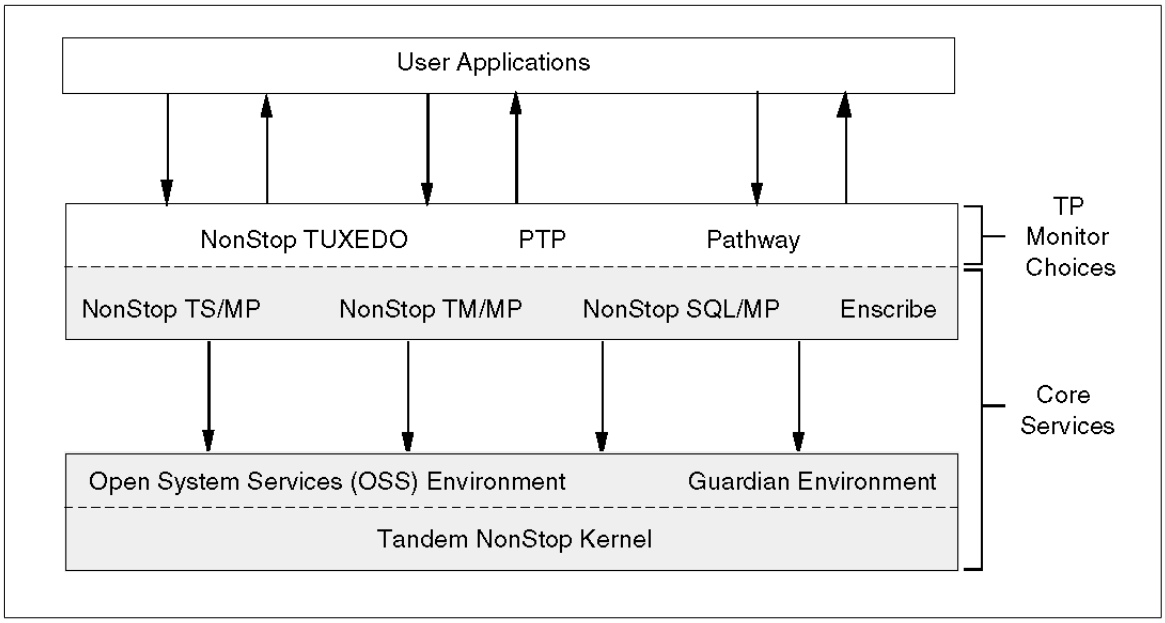


Abbildung 3-4: Einordnung von Tuxedo in das übrige System [Tand95e]

Der Transaktionsmonitor System/T sorgt für die Anbindung der Clients an die Server, ist für die Prozeß- und Transaktionsverwaltung zuständig und kümmert sich um Sicherheitsaspekte. Ziel ist die Optimierung von Durchsatz und Antwortzeit der OLTP-Anwendung.

3.2.1 Entwicklung von OLTP-Anwendungen

Mit dem Application-Transaction Monitor-Interface (ATMI) stellt System/T ein API für die Programmiersprachen C, C++ und COBOL zur Verfügung, mit dem die Entwicklung von OLTP-Anwendungen unterstützt wird. ATMI bietet unter anderem folgende Funktionalität:

- Abgrenzen von Transaktionen: Mit tpbegin(), tpcommit() und tpabort() können Transaktionen gestartet, erfolgreich beendet bzw. abgebrochen werden. Außerdem können Transaktionen mit tpsuspend() ausgesetzt und mit tpresume() wieder aufgenommen werden.
- Automatische Transaktionen: Normalerweise entscheidet der Client darüber, ob vom Server erbrachte Dienste im Transaktionsmodus ablaufen, indem er vor dem Aufruf eine Transaktion startet oder aber auch nicht. Für jeden Dienst kann festgelegt werden, daß falls beim Aufruf des Dienstes noch keine Transaktion besteht, automatisch eine Transaktion begonnen wird.
- Zwei Modelle für die Client/Server-Komunikation: Request/Response-Modell und Conversational-Modell. Diese werden weiter unten beschrieben.
- Synchrone und asynchrone Kommunikation: Diese werden ebenfalls weiter unten vorgestellt.
- Pufferverwaltung: Die Kommunikation zwischen Client und Server wird über typisierte Puffer abgewickelt. System/T bietet sieben Puffertypen, es können aber auch eigene definiert werden. ATMI bietet Funktionen zum Allokieren, Freigeben, und Anpassen von Puffern an. Außerdem wird bei einigen Puffertypen eine automatische Datenkonversion durchgeführt, falls dies beim Datenaustausch zwischen verschiedenen Plattformen, z.B. Workstations und einem Tandem NonStop System, nötig ist.

Mit dem buildclient- und buildserver-Kommando stehen dem Entwickler zwei einfach zu handhabende Befehle zur Erzeugung von Client- bzw. Serveranwendungen zur Verfügung. Buildclient kompiliert den übergebenen Client-Quellcode und bindet das Resultat mit den benötigten Tuxedo-Bibliotheken zum ausführbaren Client-Programm. Für den Server müssen nur die von diesem bereitgestellten Dienste codiert werden. Tuxedo stellt einen Serverrahmen bereit, der eingehende Aufrufe an die adäquaten Funktionen weiterleitet. Durch Überschreiben der Funktionen tpsvrinit() und tpsvrclose() können beim

Starten bzw. Beenden des Servers anstehende Aktionen durchgeführt werden. Buildserver kompiliert den Server-Quelcode und fügt Serverrahmen und Tuxedo-Bibliotheken zum ausführbaren Serverprogramm zusammen.

3.2.2 Administration einer OLTP-Anwendung

Neben der Entwicklung unterstützt Tuxedo auch die Administration einer OLTP-Applikation. Dazu stehen mehrere Werkzeuge zum Konfigurieren, Starten und Beenden und zum Beobachten einer Anwendung zur Verfügung. Neben Werkzeugen, welche über die Kommandozeile gesteuert werden, steht mit xtuxadmin auch eines mit einer grafischen Benutzeroberfläche zur Verfügung.

Die Konfiguration einer Tuxedo-Applikation erfolgt über eine ASCII-Datei, in der die beteiligten Hard- und Softwareressourcen aufgeführt werden. Zu diesen Informationen zählen u.a. die beteiligten Rechner, die Zahl der zu akzeptierenden Clients, auf welchen Rechnern welche Server und Dienste ablaufen sollen, verschiedene Charakteristiken der Server, wie z.B. Prozeßrecovery-Kriterien oder Timeout-Zeiten, Sicherheitsaspekte, verwendete Message-Queues usw.. Viele dieser Eigenschaften können dynamisch, d.h. zur Laufzeit der Anwendung, ohne diese beenden zu müssen, beeinflußt werden. So können neue Rechner, Server und Dienste bereitgestellt, andere außer Betrieb genommen oder Server auf andere Rechner verschoben werden. Timeout-Zeiten, Prioritäten und andere Parameter können verändert werden. Der Zustand der Anwendung kann abgefragt und Statistiken z.B. über die Last einzelner Rechner oder die Nachfrage bestimmter Dienste können abgerufen werden.

Das Tuxedo-System nimmt dem Administrator einen Teil der anfallenden Arbeit ab. So führt das System z.B. automatisch eine Lastverteilung durch oder startet abgestürzte Prozesse neu. Diese Tätigkeiten werden gemäß den in der Konfigurationsdatei spezifizierten Vorgaben verrichtet.

3.2.3 Kommunikationsmechanismen

Wie bereits mehrfach erwähnt, arbeiten die Komponenten einer Tuxedo-Anwendung auf der Basis des Client/Server-Modells zusammen. Im folgenden werden die zwei dafür vorgesehenen Kommunikationsformen beschrieben. Anschließend werden als weiterer Kommunikationsmechanismus die mit System/Q verfügbaren Message-Queues vorgestellt.

Das Request/Response-Modell

In diesem Modell ruft der Client einen Dienst des Servers auf und übergibt dabei die erforderlichen Parameter. Der Server führt diesen Dienst aus und liefert das Ergebnis an den Client zurück. Der Aufruf kann dabei mittels der ATMI-Funktion tpcall() synchron durchgeführt werden, d.h. der Client ist während der gesamten Ausführung des Dienstes blockiert und kann erst nach

Erhalt des Rückgabewerts weiterarbeiten. Der Aufruf kann aber durch die ATMI-Funktion tpacall() auch asynchron erfolgen, d.h. der Client kann unmittelbar nach Absetzen des Aufrufs weiterarbeiten. Während der Ausführung des Dienstes arbeiten Client und Server also parallel. Der Client kann auch weitere Server aufrufen. Um das Ergebnis eines asynchronen Aufrufs zu ermitteln, ruft der Client die ATMI-Funktion tpgetrply() auf. Falls der Server noch mit dessen Berechnung beschäftigt ist, wird ein Fehlerwert zurückgeliefert, und der Client kann es später noch einmal versuchen.

Dieses Modell eignet sich für kontextfreie Kommunikation, da der Server zustandslos ist. Außerdem spezifiziert der Client beim Aufruf eines Dienstes nicht den Server, sondern nur den Namen des gewünschten Dienstes. Das Tuxedo-System leitet den Aufruf dann an einen Server, der einen solchen Dienst anbietet weiter. Bieten mehrere Server diesen Dienst an, so wählt das System auf Grund verschiedener Kriterien, z.B. der gegenwärtigen Last, einen davon aus. Es kann also durchaus sein, daß beim Aufruf von mehreren Diensten durch den Client, auch mehrere Server zum Einsatz kommen.

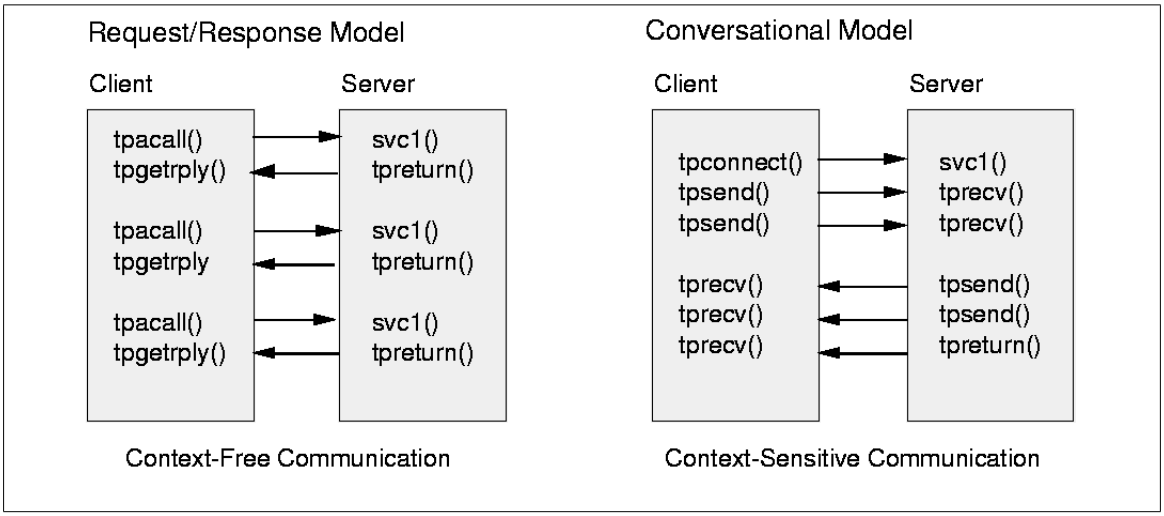


Abbildung 3-5: Kommunikationsformen für die Client/Server-Interaktion [Tand95e]

Das Conversational-Modell

Wird kontextsensitive Kommunikation benötigt, so kann auf das Conversational-Modell zurückgegriffen werden. Durch die vom Client aufgerufene ATMI-Funktion tpconnect() wird dazu explizit eine Verbindung zu einem Server aufgebaut. Diese Verbindung bleibt bis zum Ende der Konversation bestehen. Im Normalfall wird die Konversation vom Server durch Aufruf von tpreturn() beendet, sie kann aber auch vom Client durch Aufruf von tpdisconnect() terminiert werden. Während die Verbindung besteht können Client und Server Nachrichten im Halbduplex austauschen, d.h. es kann immer nur ein Partner senden. Erreicht wird dies durch ein Senderecht, das vom derzeitigen Inhaber explizit an den Partner übergeben werden muß. Gesendet wird mit der ATMI-Funktion tpsend(), Empfangen mit tprecv().

Abb. 3-5 veranschaulicht beide vorgestellten Modelle.

Message-Queues

Mit dem Konzept der Message-Queue bietet Tuxedo auch einen asynchronen Kommunikationsmechanismus an, bei dem die Kommunikationspartner nicht gleichzeitig verfügbar sein müssen. Message-Queues basieren auf benannten Warteschlangen. Diese stellen zwei Operationen zur Verfügung. Mit der queue-Operation (in anderen Systemen auch put genannt) wird die übergebene Nachricht in die angegebene Warteschlange eingereiht. Durch die dequeue-Operation (in anderen Systemen auch get genannt) kann die nächste Nachricht aus der durch ihren Namen spezifizierten Warteschlange entnommen werden. Queues sind von Natur aus passiv, d.h. sie akzeptieren queue- und dequeue-Operationen, initiieren diese aber nie von sich aus. In der Regel werden die Nachrichten nach dem FIFO-Prinzip (First In First Out) zugeteilt. Dies kann aber durch verschiedene Optionen, die weiter unten beschrieben werden, anderweitig beeinflußt werden. Durch den global bekannten Namen der Warteschlange wird Ortstransparenz erreicht. Das Konzept der Message-Queues ist sehr flexibel und erlaubt die Realisierung unterschiedlicher, auch komplexer Kommunikationsmuster. Abb. 3-6 zeigt beispielsweise wie Client/Server-Kooperation und eine Pipeline auf der Basis von Message-Queues realisiert werden können.

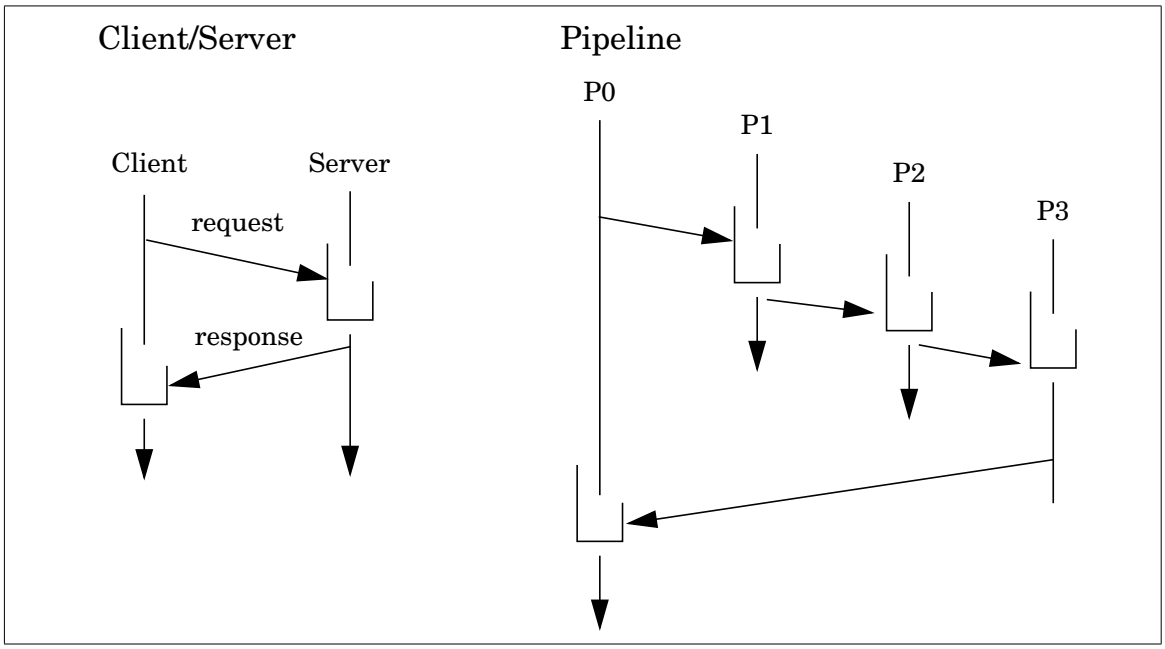


Abbildung 3-6: Client/Server und Pipeline basierend auf Message-Queues

Möchte eine Anwendung mit einer anderen kommunizieren, so legt sie mit der queue-Anweisung eine Nachricht in einer der Partneranwendung bekannten Message-Queue ab. Diese Anweisung ist nicht blockierend, und die Anwendung kann umgehend mit der Programmausführung fortfahren. Auch muß der Emp-

fänger nicht verfügbar sein. Ist der Empfänger zur Entgegennahme einer Nachricht bereit, kann er sich mit der dequeue-Anweisung die nächste Nachricht aus der Message-Queue aushändigen lassen.

Die Nachrichten einer Tuxedo-Message-Queue werden auf stabilem Speicher abgelegt, und sind daher persistent. Dadurch können sie auch Systemabstürze überdauern. Die von Tuxedo bereit gestellten Message-Queues sind außerdem transaktional. Wird im Rahmen einer Transaktion durch queue eine Nachricht in die Warteschlange eingereiht, die Transaktion aber abgebrochen (Rollback), so wird die Nachricht wieder entfernt. Die Nachricht erscheint erst in der Warteschlange, sobald die Transaktion erfolgreich abgeschlossen wurde (Commit). Analog verhält sich die dequeue-Operation. Wird im Rahmen einer Transaktion durch dequeue eine Nachricht aus der Warteschlange ausgelesen, die Transaktion aber abgebrochen, so verbleibt die Nachricht in der Warteschlange.

Wie oben erwähnt, werden Nachrichten normalerweise in der Reihenfolge ausgelesen, in der sie auch eingetragen wurden. Dieses Verhalten läßt sich durch verschiedene Parameter ändern. Bei der queue-Operation kann eine Priorität zwischen 1 und 100 angegeben werden. Nachrichten mit höherer Priorität werden beim Aufruf von dequeue vor solchen mit niedrigerer Priorität ausgelesen. Auch durch die Angabe einer sogenannten Geburtszeit, die absolut (z.B. 30. Juni 1998, 12:00) oder relativ (z.B. in 2 Stunden) angegeben werden kann, ist es möglich das FIFO-Verhalten zu beeinflussen. Nachrichten, die mit einer Geburtszeit in die Warteschlange eingetragen wurden, können erst ab dem spezifizierten Zeitpunkt ausgelesen werden. Eine weitere Option besteht darin, eine Nachricht vor allen anderen oder einer speziellen Nachricht in die Warteschlange einzutragen. Im letztgenannten Fall ist zusätzlich die Angabe der eindeutigen Nachrichten-ID der Nachricht, vor der die einzureihende erscheinen soll, erforderlich. Beim Aufruf von queue wird für die übergebene Nachricht eine eindeutige Nachrichten-ID erzeugt und an den Aufrufer zurückgegeben. Nachrichten können beim Aufruf von queue mit einer durch den Aufrufer definierten Marke (Correlation Identifier) versehen werden. Diese Marken müssen nicht eindeutig sein, können also auch mehreren Nachrichten mitgegeben werden.

Beim Entnehmen von Nachrichten aus der Warteschlange stehen drei Alternativen zur Wahl. Wird beim Aufruf von dequeue eine Nachrichten-ID angegeben, so versucht die Message-Queue die damit verbundene Nachricht auszulesen. Wird eine der oben erwähnten Marken angegeben, so wird die nächste mit dieser Marke versehene Nachricht aus der Warteschlange entfernt und zurückgegeben. Werden weder Nachrichten-ID noch Correlation Identifier angegeben, so wird die nächste Nachricht aus der Warteschlange entnommen.

Wie Abb. 3-7 verdeutlicht, ist es mit Hilfe des Correlation Identifiers möglich mehrere logische Queues innerhalb einer physischen einzurichten. Im Beispiel wird die physische Queue anhand des Correlation Identifier in drei logische Queues (NJ für New Jersey, NY für New York und Penn für Pennsylvania) partitoniert.

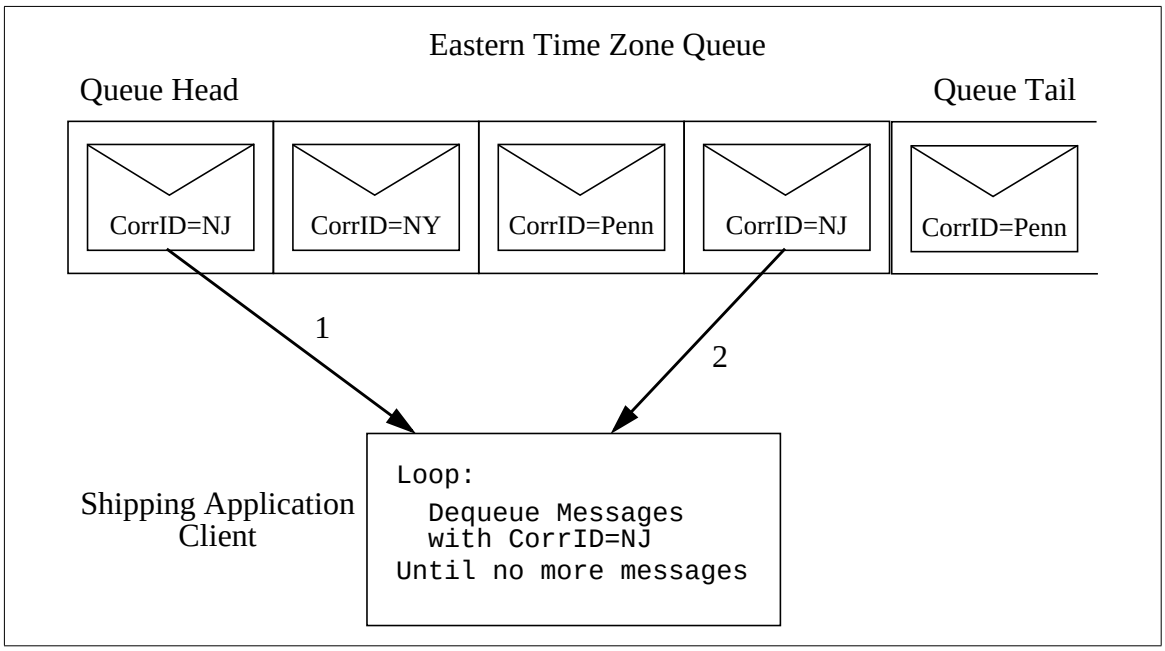


Abbildung 3-7: Partitionierung einer Message-Queue [ACDF]

3.3 Die Java-Entwicklungsumgebung

Mit dem NonStop Java Server 1.0 stand eine auf dem JDK 1.1 (Java Development Kit) von Sun basierende und an die Tandem-Himalaya-Plattform angepaßte Java-Entwicklungsumgebung zur Verfügung. Da Tuxedo für die Anwendungsentwicklung zur Zeit nur die Programmiersprachen C, C++ und COBOL vorsieht, das Agentensystem aber in Java realisiert werden sollte, mußte die Java-Anbindung aus Tuxedo-Anwendungen mit Hilfe des Invocation-API des Java-Native-Interface (JNI) erfolgen. Dieses ermöglicht einem C-Programm den Start einer JVM und die Ausführung von Java-Klassen innerhalb dieser JVM. Neben den in einem vorangegangen Abschnitt vorgestellten Kommunikationsformen, die das Tuxedo-System zur Verfügung stellt, bietet auch das JDK mit der Remote Method Invocation (RMI) einen Mechanismus für die Kommunikation zwischen den Komponenten einer verteilten Anwendung an.

3.3.1 Remote Method Invocation

RMI gehört seit dem Erscheinen des JDK 1.1 zur Java-Sprachdefinition und ist ein einfach zu handhabendes Mittel für die Entwicklung verteilter Java-Anwendungen. Mit ihr können Methoden entfernter Objekte, die sich auf anderen JVM und sogar auf anderen Rechnern befinden, aufgerufen werden. RMI beschränkt sich dabei aber auf Java-Objekte. Die Objekte, die anderen Objekten Methoden anbieten möchten, registrieren sich und damit diese Methoden unter einem festen Namen bei einem einfachen Naming-Service, welcher von der im JDK enthaltenen RMI-Registry-Applikation realisiert wird. Anwendungen, die sich für diese Methoden interessieren, wenden sich an die RMI-Registry und erhalten von dieser Informationen für den Zugriff.

Das RMI-System besteht aus verschiedenen Schichten, die in Abb. 3-8 dargestellt sind und folgende Funktionalität bereitstellen:

- Die Stub- und Skeleton-Schichten dienen als Stellvertreterobjekte in der Client- bzw. Serverapplikation, um den entfernten Zugriff für Client- und Serverimplementierung wie einen lokalen erscheinen zu lassen.
- Die Remote-Reference-Schicht sorgt für das Verpacken von Methodenaufruf und Parametern einerseits und des Rückgabewerts andererseits für den Transport über das Netzwerk.
- Die Transportschicht stellt die Netzwerkverbindung her und wickelt den Verkehr ab.

Die drei Schichten sind durch ihre Schnittstellen genau definiert und können deshalb unabhängig voneinander neu implementiert werden. So basiert die Transportschicht gegenwärtig auf TCP (Transmission Control Protocol), denkbar wäre aber auch UDP (User Datagram Protocol) oder IIOP (Internet Inter ORB Protocol).

Führt eine Client-Applikation einen entfernten Methodenaufruf durch, so wird dieser mitsamt den Argumenten via Stub, Remote-Reference- und Transportschicht über das Netzwerk auf die Serverseite übertragen. Dort wird er über Transport- und Remote-Reference-Schicht an den Skeleton und schließlich an die Serverimplementation der Methode weitergereicht. Der Rückgabewert der Methode nimmt den umgekehrten Weg zurück zum Client.

Abschließend soll das Erstellen einer RMI-Anwendung noch kurz skizziert werden. Ausgangspunkt dabei sind die Methoden, die ein Objekt anderen zur Verfügung stellen möchte. Deren Signaturen (also Typ des Rückgabewerts, Methodenname, Typen der Methodenparameter) werden in einem Remote-Interface spezifiziert, welches das Interface java.rmi.Remote erweitern muß. Letzteres enthält keine Methodenspezifikationen, sondern dient ausschließlich

dazu, den entfernten Zugriff zu signalisieren. Jede Methodendeklaration muß

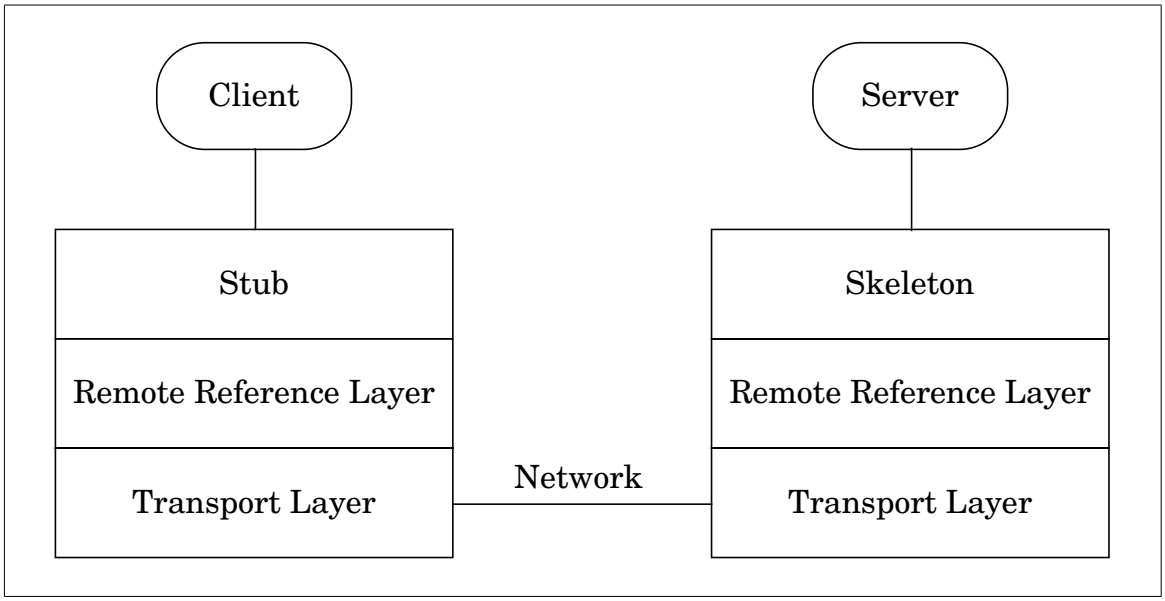


Abbildung 3-8: Das RMI-Schichtenmodell

eine throws-Klausel für java.rmi.RemoteException enthalten, mit der das RMI-System Fehler beim Aufruf einer entfernten Methode anzeigen kann. Die Klasse für das Serverobjekt, das die Methoden zur Verfügung stellt, implementiert das erwähnte Remote Interface und damit diese Methoden. Sie ist in der Regel Unterklasse von java.rmi.server.RemoteObject. In ihrer main()-Routine muß ein SecurityManager gesetzt werden. Außerdem muß in main() das Objekt bei der RMI-Registry angemeldet werden, um anderen den Zugriff darauf zu ermöglichen. Die Klasse java.rmi.Naming stellt dazu die Methoden bind() und rebind() zur Verfügung, die als Parameter eine Zeichenkette mit dem Namen, unter dem das Objekt erreichbar sein soll, erwarten. Diese Zeichenkette hat die Form eines URL (Uniform Resource Locator) und besteht aus Protokoll (rmi), Hostname, optional dem Port der RMI-Registry und dem Namen des Objektes. Wird der Port der RMI-Registry weggelassen, so wird deren Defaultwert, nämlich 1099, verwendet. Ein Beispiel eines solchen URL ist “rmi://www.myhost.de:1234/my-object”. Um auf die Methoden des Serverobjekts zugreifen zu können, muß sich der Client eine Referenz auf dieses beschaffen. Dies geschieht durch Anfrage bei der RMI-Registry des Serverrechners mittels der von java.rmi.Naming bereitgestellten lookup()-Methode, die als Parameter den URL des gewünschten Remote-Objekts erwartet. Mit dieser Referenz kann der Client auf die entfernten Methoden in der Art und Weise zugreifen, mit der er auch lokale Methoden aufruft.

Außer den class-Dateien von Client und Serverobjekt, die man mit javac erzeugen kann, benötigt man zur Ausführung noch den Bytecode für Stub und Skeleton. Diese Klassen können mit dem im Lieferumfang des JDK enthaltenen

RMI-Compilers rmic aus dem Serverobjekt generiert werden. Bevor Server und Client wie üblich mit dem java-Interpreter ausgeführt werden, muß die RMI-Registry auf dem Serverrechner mit dem Kommando rmiregistry gestartet werden. Als Parameter kann dabei der zu verwendende Port angegeben werden.

Konzeption des Agentensystems

Kapitel 4

Nachdem im letzten Kapitel die Eigenschaften der Tandem-Himalaya-Plattform und der darauf aufsetzenden Softwarekomponenten vorgestellt wurden, sollen mit den sich daraus ergebenden Möglichkeiten die in Kapitel zwei geschilderten Mängel von MOLE im neuen Agentensystem behoben werden. In diesem Kapitel wird hierzu die Konzeption des Agentensystems erläutert. Im einzelnen werden Verarbeitungsmodell, Mobilitäts- und Kommunikationskonzepte behandelt.

4.1 Verarbeitungsmodell

In diesem Abschnitt wird das Verarbeitungsmodell, das die grobe Spezifikation des Agentensystems darstellt, beschrieben. Dazu werden Architektur und die Komponenten des Agentensystems, der Lebenszyklus eines Agenten und die Probleme, die das neue System mit sich bringt, vorgestellt. Auf Kommunikation und Migration wird erst in darauf folgenden Abschnitten eingegangen.

4.1.1 Architektur des Agentensystems

Abb. 4-1 zeigt die Architektur des neuen Agentensystems, auf dessen einzelne Komponenten und deren Zusammenspiel im folgenden näher eingegangen wird.

Die Message-Queue

Als Hauptnachteil von MOLE erwies sich dessen mangelnde Zuverlässigkeit, d.h. die Möglichkeit, daß durch Amoklauf oder Absturz eines Prozesses bzw. des ganzen Systems Agenten und deren Daten verlorengehen können. Gerade auch im Hinblick auf den Einsatz von Agentensystemen im kommerziellen Umfeld ist ein solcher Verlust jedoch untragbar. Als Ursache für die mangelnde Zuverlässigkeit wurde das Fehlen von Zeitpunkten, an denen Agenten in einem persistenten Zustand abgelegt werden, festgestellt. Grundidee des neuen Agentensystems ist deshalb, die Agenten nach dem Erzeugen, einer Migration oder falls diese in einem Wartezustand geraten auf stabilem Speicher abzulegen. Wartezustände entstehen, wenn Agenten keine Berechnungen mehr

durchführen, sondern nur auf Aktionen eines Kommunikationspartners (Nachrichten, entfernte Methodenaufrufe, Etablieren einer Session) warten. Für das Speichern der Agenten kommen die im vorangegangenen Kapitel vorgestellten Message-Queues zum Einsatz, die sicherstellen, daß die darin abgespeicherten Agenten auch Systemabstürze überdauern. Ihnen wurde auf Grund des vergleichsweise einfachen Zugriffs der Vorzug vor einer Datenbank gegeben. Beim Ablegen eines Agenten in die Message-Queue wird als Correlation Identifier der Zustand des Agenten angegeben. Dieser Zustand kann entweder “zur Ausführung bereit” (nach dem Erzeugen bzw. einer Migration) oder “blockiert” (in einem Wartezustand) sein. Die Message-Queue wird mit diesem Correlation Identifier logisch in zwei Teile partitioniert, einer Warteschlange für die ausführbaren Agenten und einer für die blockierten. Prinzipiell wäre diese Trennung auch durch die Verwendung zweier Message-Queues möglich gewesen. Da dies zu einem, wenn auch geringen, Overhead geführt hätte, und die Partitionierung mit Hilfe des Correlation Identifiers sehr einfach ist, wurde auf die zweite Message-Queue verzichtet.

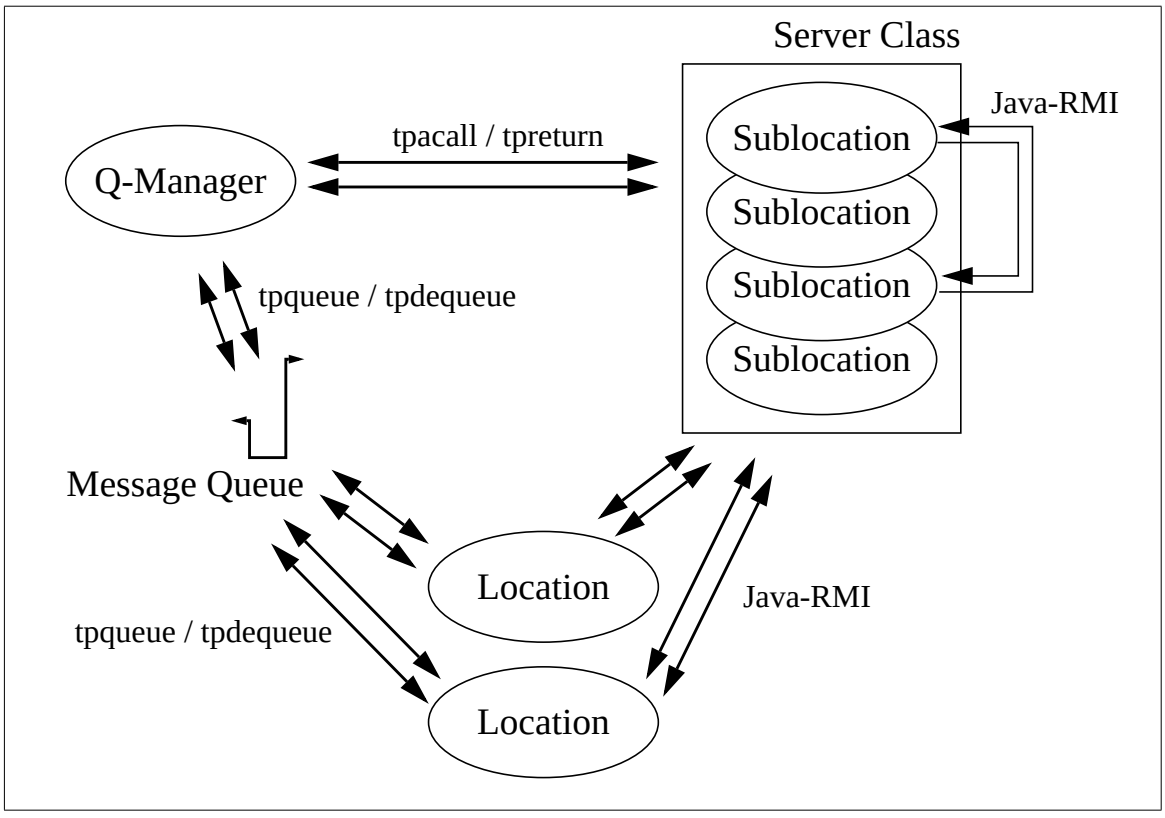


Abbildung 4-1: Architektur des Agentensystems

Wie auch aus Abb. 4-1 ersichtlich, werden Agenten sowohl von ihrem Aufenthaltsort, als auch vom Q-Manager in die Message-Queue eingetragen. Der Q-Manager schreibt nur Agenten, die in einen Wartezustand gerieten in die Message-Queue. Der Ort tut dies nach dem Erzeugen, einer Migration oder um

einen Wartezustand aufzulösen Die Message-Queue kann dabei durchaus von mehreren Orten zum sicheren Speichern der sich auf ihnen aufhaltenden Agenten benutzt werden.

Die Serverklasse

Auch die Skalierbarkeit des MOLE-Systems, die bei der kommerziellen Nutzung von Agentensystemen ebenfalls eine wichtige Rolle spielt, wurde in Kapitel zwei bemängelt. Dieser Mangel lag in der Ausführung aller Agententhreads eines oder sogar mehrerer Orte in einem einzigen Prozeß begründet. In Anbetracht der Mehrprozessorarchitektur des K10000-Rechners wäre das damit verbundene Außerachtlassen von zusätzlichen Prozessoren ziemlich unbefriedigend. Außerdem ist die Prozeßerzeugung und -verwaltung in diesem System hinreichend optimiert, so daß die Kostenvorteile von Threads gegenüber Prozessen, die bei anderen Systemen wie z.B. bei Einprozessorrechnern beträchtlich sein können, an Gewicht verlieren. Deshalb wird im neuen Agentensystem jeder Agent in einer eigenen Java-Virtual-Machine ausgeführt, die jeweils in einem eigenen Prozeß innerhalb einer Serverklasse abläuft. Eine Serverklasse ist, wie in Kapitel drei schon erläutert, eine Menge von Prozessen, die alle dasselbe Serverprogramm ausführen und kann zur Lastverteilung eingesetzt werden. Die Prozesse der Serverklasse stellen als einzigen Dienst, der auf dem Request/Response-Modell beruht, die Ausführung eines Agenten zur Verfügung. Als Parameter erwartet dieser im folgenden ExecuteAgent-Dienst genannte Dienst dabei den serialisierten Agenten, der ausgeführt werden soll. Dazu wird eine sogenannte Sublocation, die weiter unten beschrieben wird, gestartet. Diese startet ihrerseits die Ausführung des Agenten. Gerät der Agent dabei in einen Wartezustand, so wird seine Ausführung beendet und der neue Agentenzustand in serialisierter Form zurückgegeben. Beendet sich der Agent oder migriert er auf einen anderen Ort, wird dies mit einem entsprechenden Rückgabewert angezeigt.

Der Q-Manager

Ein Q-Manager genannter Prozeß sorgt dafür, daß die in der Message-Queue gespeicherten und mit dem Zustand “zur Ausführung bereit” versehenen Agenten sobald wie möglich ausgeführt werden. Da die für die Ausführung zuständige Serverklasse nur eine begrenzte Anzahl von Serverprozessen bereitstellt, ist dies nicht immer sofort nach dem Erscheinen des Agenten in der Message-Queue möglich. Wird oder ist ein Serverprozeß frei, so liest der Q-Manager den nächsten ausführbaren Agenten aus der Message-Queue und leitet dessen Ausführung durch asynchronen Aufruf des ExecuteAgent-Dienstes ein. Durch regelmäßiges Aufrufen der tpgetrply()-Funktion wird geprüft, ob die Ausführung des Agenten bereits beendet ist. Trifft dies zu, und geriet der Agent in einen Wartezustand, so wird der vom ExecuteAgent-Dienst zurückgegebene serialisierte Agent in die Message-Queue eingetragen.

Wurde die Ausführung des Agenten auf Grund seiner Termination oder einer Migration beendet, nimmt dies der Q-Manager zur Kenntnis. Die geschilderten Aktionen laufen im Rahmen einer Transaktion ab. Dieser Vorgang wird für mehrere Agenten durchgeführt. Die einzelnen Aktionen (Auslesen aus der Queue, Ausführen des Agenten, Ende der Ausführung abfragen, evtl. zurückgegebenen Agenten in die Queue schreiben) für die verschiedenen Agenten laufen dabei nicht seriell, sondern ineinander verschachtelt ab, um die Serverklasse optimal auszulasten. Dies erfordert aber ein Umschalten der beteiligten Transaktionen mittels der ATMI-Funktionen tpsuspend() und tpresume().

Beim Start der Transaktionen wird eine Timeout-Zeit angegeben. Stürzt ein einen Agenten ausführender Serverprozeß oder etwa der Q-Manager ab, so wird die Transaktion nach dieser Timeout-Zeit abgebrochen und der Agent erscheint in seinem alten Zustand wieder in der Message-Queue. Von dort kann er wieder ausgelesen und zur Ausführung gebracht werden, wobei das Agentensystem feststellt, daß der Agent einen Absturz hinter sich hat und dies dem Agenten durch Aufruf einer festgelegten Methode des Agenten signalisiert. Diese Methode kann vom Agentenprogrammierer überschrieben werden, und der Agent kann somit adäquat auf einen Absturz reagieren.

Die Ausführung des ExecuteAgent-Dienstes, und somit auch die des Agenten, läuft innerhalb der vom Q-Manager gestarteten Transaktion ab. Greift der Agent auf lokale Dienste zu, die von einem Ressource-Manager verwaltet werden (z.B. eine Datenbank), so wird dieser Dienst ebenfalls im Rahmen der Transaktion durchgeführt. Da andere Dienste, wie z.B. auch die Kommunikation mit anderen Agenten, nicht transaktional sind, kann der Absturz eines Agenten zu Problemen führen, wenn nicht adäquat auf die Signalisierung des Absturzes durch das Agentensystem reagiert wird.

Die Vorgabe der Transaktionsgrenzen durch das Agentensystem ist ungewohnt. Es wäre auch denkbar gewesen, die Ausführung des ExecuteAgent-Dienstes durch Setzen eines Flags beim tpacall()-Aufruf von der Transaktion auszuschließen, und es dem Agentenprogrammierer zu erlauben, selbst Transaktionsgrenzen anzugeben. Dies hätte aber zu dem Problem geführt, daß im Falle eines Agentenabsturzes die bisher erfolgreich abgeschlossenen und dauerhaft gewordenen Transaktionen beim Wiederausführen des Agenten wiederholt, und damit ein weiteres mal durchgeführt würden. Um dem Programmierer das Begrenzen von Transaktionen gestatten zu können, würden verschachtelte (nested) Transaktionen benötigt. Die bei der Ausführung des Dienstes vom Agenten durchgeführten Transaktionen würden nur dauerhaft, wenn die den ExecuteAgent-Dienst umfassende Transaktion erfolgreich abgeschlossen würde. Tuxedo bietet aber nur das flache (flat) Transaktionsmodell an.

Wartezustände

Wie schon erwähnt, steht nur eine begrenzte Zahl von Serverprozessen, die den ExecuteAgent-Dienst anbieten, zur Verfügung. Deshalb ist es sinnvoll, daß Agenten, die gerade ausgeführt werden, dabei aber in einen Wartezustand geraten, den ExecuteAgent-Dienst verlassen und in die Message-Queue ausgelagert werden. Löst sich der Wartezustand auf, so können die Agenten wieder zur Ausführung kommen. Ein Wartezustand entsteht dann, wenn der Agent, der auch mehrere Ausführungsstränge (Threads) besitzen kann, keine Berechnungen mehr durchführt, sondern nur auf den Empfang einer Nachricht, den Eingang eines entfernten Methodenaufrufs, die Etablierung einer Session oder die Meldung eines Weckdienstes nach Aufruf einer Sleep-Funktion wartet.

Bei der Umsetzung der Auslagerung steht das Agentensystem vor dem Problem einen Wartezustand zu Erkennen, und falls dies glückt, den kompletten Ausführungszustand des Agenten zu ermitteln, so daß dieser nach der Auflösung des Wartezustands wiederhergestellt werden kann. Letztgenanntes Problem tritt auch bei der Migration eines Agenten auf und ist nicht einfach zu lösen, da dies Eingriffe in die Java-Virtual-Machine erfordern würde. Im Vorgriff auf den Abschnitt über die Migration sei gesagt, daß dieses Problem umgangen wird, indem eine “schwache” Migration zum Einsatz kommt. Analog wird auch nach Auflösung eines Wartezustands agiert, indem nicht der alte Ausführungszustand restauriert wird, sondern nur auf das Ereignis, das den Wartezustand beendete, reagiert wird. Zum Beispiel wird im Falle eines Nachrichtenempfangs nur die dafür vorgesehene Methode des Agenten aufgerufen. Auch das Erkennen eines Wartezustands bereitet Schwierigkeiten, und das Agentensystem verläßt sich deshalb auf die Kooperation der Agenten bzw. deren Programmierer. Wird auf ein Ereignis gewartet, so sollte dies nicht durch Polling bzw. dadurch geschehen, daß der Agententhread einfach schlafen gelegt wird. Vielmehr sollte der Agententhread beendet oder die bereits erwähnte Sleep-Funktion aufgerufen werden. Diese beendet den Thread implizit, sorgt aber dafür, daß nach Verstreichen der angegebenen Zeitspanne eine festgelegte Agentenmethode aufgerufen wird, so daß der evtl. in die Message-Queue ausgelagerte Agent wieder zur Ausführung kommen kann. Agenten können mehrere Ausführungsstränge besitzen und werden daher nur ausgelagert, wenn sich alle Ausführungsstränge in einem Wartezustand befinden und nach obigem Muster vorangegangen wird.

Locations und Sublocations

Orte (Locations) sind, wie auch in MOLE, Plätze, auf denen sich Agenten logisch aufhalten können. Sie dienen weiterhin zum Erzeugen von Agenten, als Ziel von Migrationen, als Anbieter von Diensten, zum Verwalten der sich auf ihnen aufhaltenden Agenten und nicht zuletzt zum Adressieren von Agenten, mit denen man kommunizieren möchte. Da aber jeder Agent in seiner eigenen JVM und somit getrennt von seinem Ort ausgeführt wird, muß ein Teil der bis-

her vollständig vom Ort erbrachten Funktionalität in die Prozesse, in denen die Agenten ablaufen, verlagert werden. Dieser Teil der Funktionalität, zu der die Bereitstellung der Kommunikations- und Migrationsprimitive zählt, wird von sogenannten Sublocations erbracht. Eine Sublocation stellt eine Art Laufzeitumgebung dar, die zu Beginn jeder ExecuteAgent-Dienstausführung gestartet wird. Agenten interagieren mit ihrer Sublocation, wie sie dies mit ihrem Aufenthaltsort tun würden. Die Sublocation reicht, falls dies erforderlich ist, diese Anfragen transparent für den Agenten an dessen Ort durch. Die Sublocation verbirgt also die Zweiteilung und verhält sich aus Sicht des Agenten wie sein Aufenthaltsort.

Zwischen Ort und den Sublocations seiner Agenten herrscht eine rege Kommunikation in beide Richtungen. Das Request/Response- und das Conversational-Modell sind dafür ungeeignet, da sie beispielsweise den gleichzeitigen Zugriff zweier Sublocations auf ihren Ort nicht gestatten würden. Denkbar wäre eine Kommunikation über eine Message-Queue gewesen. Dies hätte aber das “Pollen” der Warteschlange, also das ständige Abfragen auf neue Nachrichten, erforderlich gemacht. Nicht zuletzt auf Grund der einfachen Implementierbarkeit wurde deshalb auf die im letzten Kapitel vorgestellte Java RMI zurückgegriffen. Da eine Kommunikation in beide Richtungen benötigt wird müssen sich sowohl der Aufenthaltsort, als auch die Sublocations bei der RMI-Registry anmelden. Der Aufenthaltsort tut dies beim Start unter seinem logischen Namen. Die Sublocations melden sich zu Beginn der Ausführung des ExecuteAgent-Dienstes unter den Namen der Agenten, die sie ausführen, an.

Da Agenten, die sich in einem Wartezustand befinden, in die Message-Queue ausgelagert werden können, ist die Kommunikation zwischen Agenten mit einigen Schwierigkeiten behaftet. Zum einen gibt es keine feste Adresse, unter welcher der Kommunikationspartner ständig direkt ansprechbar ist, zum anderen kann sich der gewünschte Partner gerade in der Message-Queue befinden und muß deshalb erst zur Ausführung gebracht werden. Der die Kommunikation initiierende Agent wendet sich mit seinem Kommunikationswunsch daher an den Aufenthaltsort des Partneragenten. Falls der Partneragent blokkiert ist, löst der Ort diesen Wartezustand auf, indem er den Agenten im Rahmen einer Transaktion aus der Message-Queue ausliest und mit dem Zustand “zur Ausführung bereit” sofort wieder hineinschreibt. Der Ort wartet dann bis der Agent über den Q-Manager, wie oben beschrieben, an den ExecuteAgent-Dienst übergeben und ausgeführt wird. Der Ort erhält davon durch die Anmeldung des Agenten durch dessen Sublocation Kenntnis. Dabei wird dem Ort der RMI-URL, unter dem die Sublocation und damit der Agent erreichbar ist, übergeben und in den Datenstrukturen des Ortes gespeichert. Je nach Kommunikationsform wird dann die eigentliche Kommunikation über den Ort (asynchroner Nachrichtenversand) oder direkt zwischen den Agenten (synchroner Nachrichtenversand und entfernter Methodenaufruf) durchgeführt. Im letztgenannten Fall gibt der Ort den RMI-URL des gewünschten Partners an den die Kommu-

nikation initiierenden Agenten zurück, der sich dann mit Hilfe dieses RMI-URL über die RMI-Registry eine Referenz der Sublocation des Partner beschaffen und sich diesem direkt zuwenden kann. Dieser ganze Vorgang ist relativ ineffizient. Um eine gewisse Verbesserung zu erreichen, wird von jeder Sublocation ein Cache verwaltet, der Agentennamen auf die zuletzt gültige Referenz ihrer Sublocations abbildet. Liegt ein Kommunikationswunsch vor, so wird zuerst der Cache konsultiert. Findet sich darin eine Referenz der Sublocation des Partneragenten, so wird versucht direkt mit dieser, also ohne Umweg über den Aufenthaltsort des Partners, Kontakt aufzunehmen. Schlägt dies fehl, weil die Referenz veraltet ist, muß der beschriebene Weg über den Ort des Partners eingeschlagen werden.

4.1.2 Lebenszyklus eines Agenten

Um ein besseres Verständnis des neuen Agentensystems zu ermöglichen, soll das Zusammenspiel seiner verschiedenen Komponenten auch aus der Sicht eines einzelnen Agenten beschrieben werden.

Ein Agent kann durch Eingabe des new-Befehls (Syntax: ‘new <Agentenklasse>(<Tag> <Parameter>, <Tag> <Parameter>, ...);’) aus der Kommandozeile eines Ortes oder auch programmgesteuert bei der Ausführung anderer Agenten durch Aufruf einer vom Ort angebotenen Methode erzeugt werden. Nach dem Erzeugen wird Name und RMI-URL des Ortes im Agenten durch den Ort gesetzt und der Agent mit dem Zustand “zur Ausführung bereit” in die Message-Queue geschrieben. Außerdem werden einige Zustandsinformationen in die Liste der sich auf dem Ort aufhaltenden Agenten eingetragen. Wird ein Serverprozeß der Serverklasse frei, so startet der Q-Manager eine neue Transaktion, liest den Agenten aus der Message-Queue und leitet dessen Ausführung durch asynchronen Aufruf des ExecuteAgent-Dienstes der Serverklasse ein. Bei der Ausführung dieses Dienstes wird eine Sublocation gestartet. Diese meldet sich unter dem Agentennamen bei der lokalen RMI-Registry an, um an den Agenten gerichtete Nachrichten, entfernte Methodenaufrufe oder Sessionaufbau bzw. -abbau bearbeiten zu können. Weiterhin meldet sie den Agenten bei seinem Aufenthaltsort an. Dieser aktualisiert seine den Agenten betreffenden, internen Daten und liefert einen Wert zurück, der angibt wie weiter zu verfahren ist. Beim erstmaligen Anmelden, also nach dem Erzeugen oder einer Migration, wird dies der Aufruf der start()-Methode des Agenten sein. Diese ist agentenspezifisch und implementiert die gewünschte Agentenfunktionalität. In ihr können unter anderem Nachrichten versandt, entfernte Methodenaufrufe durchgeführt, Sessions eingegangen und eine Migration angestoßen werden. Auf diese Aktionen wird in den folgenden Abschnitten näher eingegangen.

Kommt der Agent zu dem Entschluß sich zu beenden, so kann er dies durch Aufruf einer die()- Methode tun. Diese sorgt dafür, daß der Agent aus den Datenstrukturen seines Aufenthaltsortes entfernt, die Sublocation bei der loka-

len RMI-Registry abgemeldet und anschließend beendet wird. Der Execute-Agent-Dienst wird dann mittels tpreturn() verlassen. Der Q-Manager, der regelmäßig durch Aufruf von tpgetrply() auf das Ende des Dienstes wartet, beendet schließlich die Transaktion (Commit).

Falls der Agent sich zwar nicht beenden möchte, aber nur auf den Empfang einer Nachricht oder eines entfernten Methodenaufrufs wartet, so kann er alle laufenden Threads beenden. Die Sublocation sorgt dann dafür, daß der Agentenzustand am zugehörigen Ort auf “blockiert” gesetzt wird. Außerdem meldet sich die Sublocation bei der lokalen RMI-Registry ab und beendet sich. Der ExecuteAgent-Dienst wird danach mittels tpreturn() verlassen. Der Q-Manager wird durch den übergebenenen Rückgabewert veranlaßt, den Agenten mit dem Zustand “blockiert” in die Message-Queue zu schreiben. Anschließend wird die Transaktion geschlossen (Commit).

Liegt später eine Nachricht bzw. ein entfernter Methodenaufruf für den Agenten vor, kann eine Session auf- oder abgebaut werden oder ist die beim Aufruf der Sleep-Funktion angegebene Zeitspanne verstrichen, so veranlaßt der Ort die Aktivierung des Agenten, indem er ihn aus der Message-Queue liest und mit dem Zustand “zur Ausführung bereit” sofort wieder hineinschreibt. Sobald ein Server der Serverklasse frei wird bzw. ist, wird die Ausführung des Agenten durch den Q-Manager, wie oben beschrieben, angestoßen. Beim Empfang einer Nachricht wird die agentenspezifische receiveMessage()-Methode, bei einem entfernten Methodenaufruf die gewünschte, beim Auf- oder Abbau einer Session die sessionEstablished()- bzw. sessionTerminated()-Methode und bei der Rückkehr nach einem Sleep-Aufruf die wakeUp()- Methode des Agenten ausgeführt..

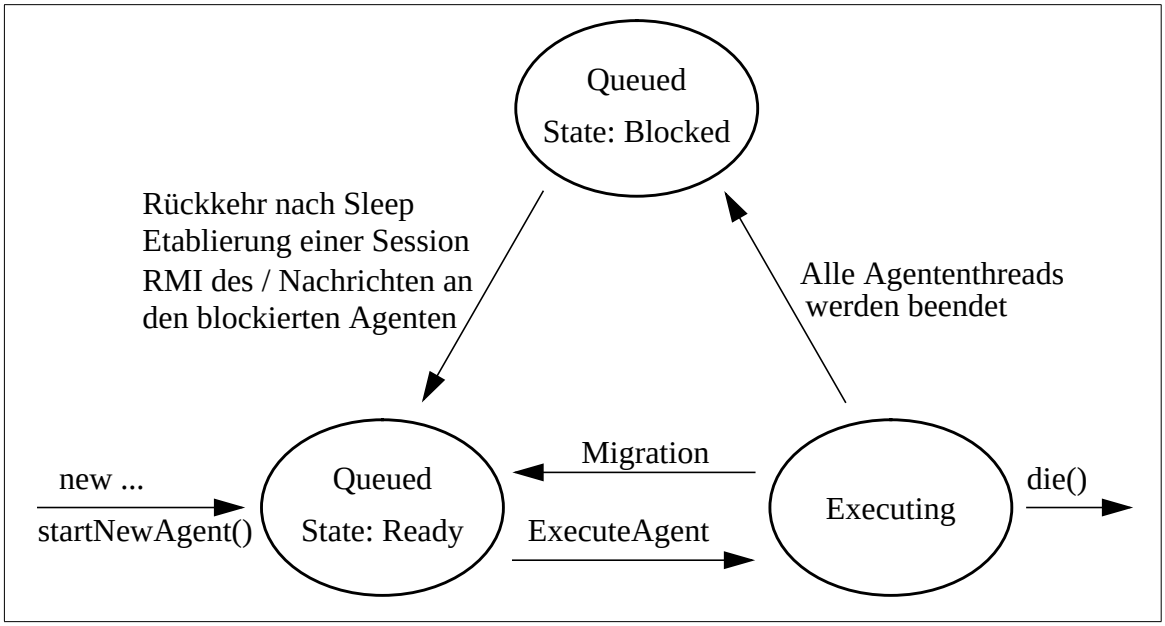


Abbildung 4-2: Lebenszyklus eines Agenten

Abb. 4-2 veranschaulicht noch einmal den Lebenszyklus eines Agenten anhand eines Zustand-Übergangsdiagramms.

4.1.3 Probleme des neuen Agentensystems

Neben den Vorteilen der höheren Zuverlässigkeit und der besseren Skalierbarkeit birgt die Konzeption des neuen Agentensystems leider auch einige Probleme, die im folgenden beschrieben werden.

Zwar können durch die Verwendung einer Message-Queue keine Agenten mehr verloren gehen, aber unproblematisch ist der Absturz eines Agenten trotzdem nicht. Abb. 4-3 zeigt dies an einem Beispiel. Agent X schickt Agent Y eine Nachricht und wartet danach auf eine Antwort von Y. Agent Y hingegen wartet auf eine Nachricht von X, um diese dann zu beantworten. Stürzt nun aber Agent Y nach Erhalt der Nachricht von X, aber noch vor dem Versand der Antwort an X ab, so entsteht nach der Recovery ein Deadlock, bei dem beide Agenten auf Nachrichten der jeweiligen Gegenseite warten. Ähnliche Probleme können nicht nur beim Nachrichtenaustausch, sondern allgemein auch beim Zugriff auf Ressourcen entstehen. Abgestürzte Agenten werden zwar über den Absturz informiert, können diese Meldung aber ignorieren und als “Speicherleichen” dann Ressourcen der Message-Queue belegen. Außerdem ist bei komplexeren Aufgaben die Art und Weise, wie auf einen Absturz reagiert werden soll, für den Agenten nicht so einfach ersichtlich. Der Agent weis zwar, daß er abgestürzt ist, vollständige Informationen über die bis dahin durchgeführten Aktionen hat er in der Regel aber nicht.

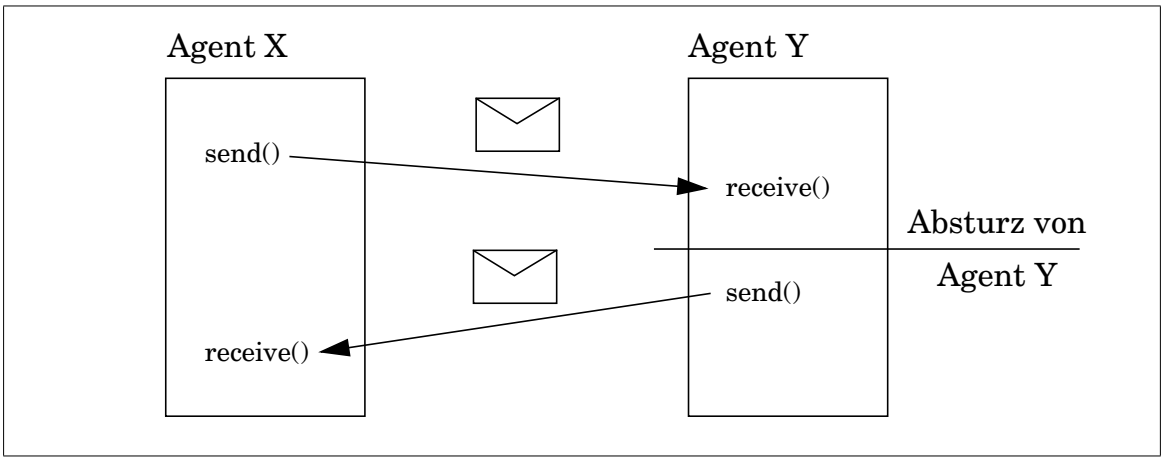


Abbildung 4-3: Zustandekommen eines Deadlocks

Bei der Erkennung von Wartezuständen ist das Agentensystem, wie schon erläutert, auf die Kooperation des Agenten bzw. dessen Programmierers angewiesen. Hält sich dieser nicht an die Vorgaben, so blockiert der Agent den Serverprozeß, der ihn ausführt. Während diese Ressourcenvergeudung schon ärgerlich ist, sich aber bei sinnvoll programmierten Agenten noch in Grenzen hält, bereiten fehlerhaft programmierte Agenten oder welche, die diesen

Umstand mutwillig ausnutzen, ernsthafte Schwierigkeiten. Gerät ein Agent z.B. in eine Endlosschleife, so blockiert er den ausführenden Serverprozeß solange, bis sein Fehlverhalten vom Administrator entdeckt und dieser von ihm aus dem System entfernt wird. Auch das mutwillige Ausnutzen dieser Sicherheitslücke ist durch sogenannte Denial-of-Service-Angriffe denkbar und möglich, indem Agenten, die solche Endlosschleifen enthalten, bewußt ins Agentensystem eingebracht werden.

```
void start() {

Aktion1;
Thread.currentThread().sleep();
Aktion2;
Thread.currentThread().sleep();
Aktion3;
Thread.currentThread().sleep(;)
Agent beenden;

}
```

```
int state;

void start() {
  Aktion1;
  state=1;
  sleep();
}

void wakeUp() {
if (state==1) {
  Aktion2;
  state=2;
  sleep();
}
else if (state==2) {
  Aktion3;
  state=3;
  sleep();
}
else
  Agent beenden;
}
```

Listing 4-1: Beispiel für die Umstellung des Programmierstils

Unterstützt der Agentenprogrammierer das Agentensystem bei der Erkennung von Wartezuständen, indem er in Wartesituationen Agententhreads explizit oder implizit, z.B. durch die Sleep-Funktion, beendet, so daß der Agent ausgelagert werden kann, erfordert dies vom Programmierer eine Änderung seines gewohnten “seriellen” Programmierstils. Listing 4-1 soll dies an einem Beispiel verdeutlichen. Die linke Seite zeigt die übliche Programmierung eines Agenten, der verschiedene Aktionen ausführt und sich zwischen diesen immer wieder schlafen legt. Die rechte zeigt eine Realisierung, die es dem System ermöglicht den Agenten in den Wartephasen auszulagern. Zum Einsatz kommt dabei die besprochene Sleep-Funktion. Ist die angegebene Zeitspanne verstrichen, so wird der Agent wieder eingelagert, und die wakeUp()-Funktion des Agenten wird aufgerufen. Der Ausführungszustand des Agenten muß manuell in den Agentendaten (die Variable state) codiert werden. Dies war auch schon im MOLE-System vonnöten, dort allerdings auf die Migration beschränkt.

Die beiden zuletzt aufgeführten Probleme lassen sich ganz oder wenigstens teilweise dadurch lösen, daß beim Auslagern in die Message-Queue der vollständige Ausführungszustand des Agenten ermittelt und beim Einlagern wiederhergestellt wird. Außerdem könnten die Serverprozesse im Zeitscheibenverfahren an die ausführbaren Agenten vergeben werden, ähnlich der Prozeßzuteilung in einem Betriebssystem. Auch die Umstellung auf "starke" Migration wäre möglich. Im Ausblick wird diese Lösungsmöglichkeit noch einmal aufgegriffen.

4.2 Kommunikationsmechanismen

Nachdem die Konturen des neuen Agentensystems vorgestellt wurden, beschäftigt sich dieser Abschnitt mit einem der grundlegendsten Dienste, den das Agentensystem einem Agenten zur Verfügung stellt, nämlich der Kommunikation mit anderen Agenten. Im folgenden werden die in MOLE realisierten Kommunikationskonzepte erläutert. Diese Beschreibung beruht auf den Darstellungen in [BaHoSt] und [BHRS]. Da das neue Agentensystem in Bezug auf die Kommunikation konzeptionell möglichst eng an MOLE angelehnt sein sollte, sind die vorgestellten Konzepte als Anforderungen zu verstehen, die das neue System erfüllen muß. Neben einer verbindungslosen Kommunikation in Form von Nachrichtenversand und entferntem Methodenaufruf stellt das Agentensystem auch eine verbindungsorientierte Kommunikation in Form von Sessions zur Verfügung. Die erwähnten Konzepte können sowohl zur "lokalen" Kommunikation, d.h. zwischen Agenten, die auf demselben Ort residieren, als auch zur globalen Kommunikation, d.h. zwischen Agenten, die sich auf verschiedenen Orten aufhalten, eingesetzt werden. Wie aus den vorangegangenen Abschnitten ersichtlich, kann es bei der Kommunikation zu Verzögerungen kommen, wenn sich der Kommunikationspartner z.B. in der Message-Queue befindet und gegenwärtig kein freier Serverprozeß zu dessen Ausführung bereitsteht. Deshalb muß bei jedem Kommunikationswunsch eine Timeout-Zeit angegeben werden. Verstreicht diese Zeitspanne, ohne daß eine Kommunikation stattfinden konnte, so wird der Kommunikationsversuch abgebrochen und der initiierende Agent darüber informiert.

4.2.1 Identifikation des Kommunikationspartners

Bevor die verschiedenen Kommunikationskonzepte beschrieben werden soll im folgenden erst geklärt werden, wie der Kommunikationspartner identifiziert wird.

Global eindeutige Agentennamen

Eine Möglichkeit der Identifizierung besteht in der Verwendung von global eindeutigen Agentennamen. Diese werden vom Agentensystem vergeben und bleiben während der gesamten Lebenszeit des Agenten unverändert. Zur Zeit ist es

in MOLE nicht möglich mit Hilfe des Agentennamens den aktuellen Aufenthaltsort des Agenten zu ermitteln. Deshalb muß bei der Spezifikation des Kommunikationspartners neben dem Agentennamen zusätzlich der Aufenthaltsort des Agenten angegeben werden. Für die Zukunft ist die Implementation eines Mechanismus geplant, der aus dem Agentennamen den aktuellen Aufenthaltsort des Agenten ableiten kann. Dabei enthält der Agentenname die Adresse seines Heimatortes, also des Ortes auf dem er erzeugt wurde. Orte speichern die aktuellen Aufenthaltsorte der auf ihnen erzeugten Agenten. Diese Angaben werden ständig aktualisiert, indem der Heimatort des Agenten bei jeder Migration über den neuen Aufenthaltsort informiert wird.

Die Spezifikation des Kommunikationspartners durch Agentennamen bietet sich bei Serviceagenten an, wenn man deren Namen durch ein Dienstverzeichnis, das Dienstnamen auf Agentennamen abbildet, ermitteln kann. Auch wenn Agenten programmgesteuert andere Agenten erzeugen, können deren Namen ermittelt und zur Kommunikation eingesetzt werden. Im allgemeinen ist die Identifikation des Kommunikationspartners über seinen Agentennamen aber relativ inflexibel.

Badges

Eine andere, flexiblere Art der Identifikation stellen sogenannte Badges dar. Badges sind Marken, die sich Agenten anheften können, um anderen Agenten vorhandene Eigenschaften, wie z.B. angebotene Dienste oder eine Gruppenzugehörigkeit, zu signalisieren. Badges müssen nicht eindeutig sein, mehrere Agenten können sich die gleiche Badge anheften. Agenten haben sogar die Möglichkeit, sich eine Badge mehrmals anzuheften. Wird beim Kommunikationswunsch eine Badge zur Identifikation des Partners angegeben, so wählt das Agentensystem, genauer der wie bei Adressierung durch Agentennamen zwingend anzugebende Ort des Partneragenten, aus den Agenten, die sich auf diesem Ort aufhalten und diese Badge tragen, einen aus. Aus Lastverteilungsgründen kommt dabei ein Round-Robin-Verfahren zum Zuge, um zu vermeiden, daß immer derselbe Agent gewählt wird. Durch die Möglichkeit eines Agenten sich eine Badge mehrmals anzuheften, wird dieser dann auch entsprechend häufig ausgewählt. Badges müssen nach dem Anheften nicht für immer getragen werden, sondern können vom Agenten auch wieder abgelegt werden.

Das Konzept der Badges soll an einem kleinen Beispiel veranschaulicht werden. Auf Ort A befinden sich Agenten, die einer Agentenarbeitsgruppe angehören, und dies durch das Tragen der Badge “AAG” kennzeichnen. Möchte ein anderer Agent mit einem Mitglied der Arbeitsgruppe kommunizieren, so gibt er als Kommunikationspartner das Paar (Ort X, Badge “AAG”) an. Das Agentensystem wählt aus den passenden Agenten dann einen aus. Um diese Kommuni-

kation mittels der Identifikation über Agentennamen durchzuführen, müßte der initiierende Agent die vom System vergebenen Agentennamen aller Mitglieder der Gruppe ermitteln können und sich dann daraus einen auswählen.

4.2.2 Entfernter Methodenaufruf

Nachdem nun die beiden in MOLE realisierten Möglichkeiten zur Identifikation eines Kommunikationspartners vorgestellt wurden, wenden sich dieser und die folgenden Abschnitte der eigentlichen Kommunikation zu. Als erstes soll der entfernte Methodenaufruf (RMI), das objektorientierte Pendant zum entfernten Prozeduraufruf (RPC), vorgestellt werden. Der übliche entfernte Methodenaufruf ist ein synchroner Kommunikationsmechanismus, bei dem durch den Aufruf einer Methode eines entfernten Objekts der Kontrollfluß und die benötigten Parameter vom aufrufenden an das aufgerufene Objekt übertragen werden bis ein Ergebnis vorliegt und dieses an den Aufrufer zurückgegeben wird.

Bezogen auf das Agentensystem bedeutet dies, daß Agenten die öffentlichen Methoden anderer Agenten aufrufen können. Der Agententhread, der den Aufruf durchführt, bleibt solange blockiert, bis der Methodenaufruf abgearbeitet ist und das Ergebnis zurückgeliefert wird. Andere Threads des Aufrufers werden nicht blockiert und laufen parallel zur entfernten Methode ab. Während ein entfernter Methodenaufruf durchgeführt wird, dürfen weder aufrufender noch aufgerufener Agent migrieren.

Der entfernte Methodenaufruf eignet sich sehr gut für die Kommunikation von Agenten mit Serviceagenten, da diese in der Regel auf einer Client/Server-Beziehung basiert. Aber auch für die Kommunikation von Agenten untereinander kann dieser Ansatz eingesetzt werden.

4.2.3 Nachrichtenaustausch

Besser geeignet ist in diesem Fall aber die Kommunikation über Nachrichten. Der Nachrichtenaustausch ist ein sehr flexibler Kommunikationsmechanismus, mit dem auch höhere Protokolle wie z.B. KQML/KIF oder der entfernte Methodenaufruf realisiert werden können. In einer früheren Version von MOLE wurde beispielsweise der RPC mit Hilfe von Request- und Result-Nachrichten implementiert. Der Nachrichtenaustausch kann asynchron oder synchron erfolgen.

Asynchroner Nachrichtenversand

Beim asynchronen Versenden von Nachrichten nimmt die Sendeanweisung die übergebenen Nachrichtenparameter entgegen und gibt den Kontrollfluß unmittelbar danach zurück. Der Nachrichtenversand wird dann parallel durchgeführt. Tritt ein Fehler auf, d.h. die Nachricht kann dem Empfänger nicht

zugestellt werden, so wird anhand der beim Aufruf der Sendeanweisung spezifizierten Fehlersemantik weiterverfahren. Das Agentensystem stellt dazu drei Möglichkeiten zur Wahl:

- Es wird nichts getan.
- Es wird eine Fehlernachricht an den Sender geschickt. Diese enthält nähere Angaben zur Fehlerursache und wird mit erstgenannter Fehlersemantik versandt.
- Die Nachricht wird am Aufenthaltsort des Empfängers gespeichert. Der Empfänger kann durch Aufruf einer vom Ort angebotenen Methode die an ihn gerichteten Nachrichten später abholen.

Synchroner Nachrichtenversand

Beim synchronen Versenden von Nachrichten bleibt der Sender solange blokkiert, bis der Empfänger die Nachricht entgegengenommen hat. Dies ist zwar weniger flexibel als der asynchrone Nachrichtenversand, da die Parallelität einschränkt wird, hat aber den Vorteil der einfacheren Handhabung durch den Agentenprogrammierer. Dieser kann nämlich davon ausgehen, daß nach Rückkehr aus der Sendeanweisung dem Empfänger die Nachricht vorliegt. In MOLE wird im Falle eines Fehlers in Abhängigkeit von der spezifizierten Fehlersemantik, wie oben beschrieben, vorgegangen. Diese indirekte Fehlerbenachrichtigung macht die Fehlerbehandlung durch den Agentenprogrammierer aber unnötig kompliziert und ist deshalb im Falle des synchronen Nachrichtenaustauschs unangebracht. Im neuen Agentensystem werden Fehler daher durch Ausnahmen (Exceptions), die von der Sendeanweisung ausgelöst und vom Agenten ausgewertet werden können, angezeigt.

Nachrichtenempfang

Nach dem Versand von Nachrichten soll nun das Gegenstück, nämlich deren Empfang, näher erläutert werden. MOLE weicht hier vom üblichen Nachrichtenparadigma ab, wonach die Adressaten von Nachrichten über deren Empfang durch Aufruf einer entsprechenden Empfangsmethode selbst bestimmen. In MOLE geschieht der Empfang von Nachrichten jedoch ereignisorientiert. Der Aufenthaltsort des Empfängeragenten ruft eine von diesem bereitgestellte Methode auf, sobald eine Nachricht vorliegt. In der Defaultimplementierung dieser Methode, die vom Agentenprogrammierer überschrieben werden kann, wird die Nachricht in eine Liste eingetragen. Durch das ständige Abfragen dieser Liste auf eingegangene Nachrichten (Polling) kann das übliche Empfangsverhalten simuliert werden.

4.2.4 Sessions

Mit den bisher vorgestellten Konzepten, also entferntem Methodenaufruf und Nachrichtenversand, sind Agenten in der Lage ohne expliziten Aufbau einer Kommunikationsbeziehung miteinander zu kommunizieren. Mit Sessions steht den Agenten aber auch ein Konzept zur Verfügung, das diesen expliziten Aufbau zwingend erfordert. Eine Session ist eine Kommunikationsbeziehung zwischen genau zwei Agenten mit folgenden Eigenschaften:

- Eine Session läßt sich in drei Phasen gliedern: Aufbau einer Session, Interaktion der beiden beteiligten Agenten, Abbau der Session
- Die beteiligten Agenten müssen sich nicht unbedingt auf demselben Ort aufhalten, sondern können sich durchaus auf verschiedenen Orten befinden.
- Agenten, die an einer Session teilnehmen sollen, müssen dies explizit kundtun.
- Agenten, die in eine Session involviert sind, können diese jederzeit auch ohne Zustimmung des Partners verlassen.
- Während einer Session dürfen die beteiligten Agenten nicht migrieren. Tun sie dies doch, so wird die Session implizit beendet. Diese Eigenschaft dient lediglich dazu, die Verwaltung von Sessions durch das Agentensystem zu vereinfachen.

Sessions bieten die Möglichkeit, kooperationswillige Agenten zu synchronisieren. Agenten können angeben mit welchen anderen Agenten sie auf welchem Ort eine Zusammenarbeit eingehen möchten. Sessions erlauben ihnen auf die Ankunft des Kooperationspartners und dessen Bereitschaft zur Kooperation zu warten. Ein weiterer Grund für die Bereitstellung von Sessions ist die Unterstützung von zustandsbehafteten Interaktionen durch das Agentensystem. Beispielsweise sind Systemagenten, die anderen Agenten die Dienste eines zustandsverändernden Servers zugänglich machen sollen, auf eine explizite Kommunikationsbeziehung, wie sie von Sessions unterstützt wird, angewiesen.

Die oben aufgeführten drei Phasen sollen nun näher beschrieben werden.

Aufbau einer Session

Das Agentensystem bietet Agenten zwei Möglichkeiten zum Aufbau einer Session. Mit ActiveSetUp können Sessions initiiert werden. Diese Anweisung ist blockierend. Erst wenn eine Session etabliert werden konnte oder die angegebene Timeout-Zeitspanne verstrichen ist, kehrt ActiveSetUp zurück. Mit PassiveSetUp erklärt der Agent lediglich seine Bereitschaft an einer Session teilzunehmen, initiiert diese aber nicht. Um eine Session aufzubauen, muß also mindestens einer der beiden Agenten ActiveSetUp durchführen. Sowohl bei ActiveSetUp, als auch bei PassiveSetUp wird der gewünschte Partner über den Agentennamen oder eine Badge spezifiziert. Eine Session kommt nur zustande,

wenn die angegebenen Partnerspezifikationen zueinander passen. Im Falle einer Badge wird aus den in Frage kommenden Agenten ein Agent ausgewählt. Die beteiligten Agenten werden durch Aufruf einer festgelegten Methode über das Zustandekommen einer Session informiert. Wie in Abb. 4-4 zu sehen, können mit PassiveSetUp und ActiveSetUp verschiedene Interaktionsmuster, wie Client/Server oder Peer-to-Peer, realisiert werden.

Im neuen Agentensystem wird der ActiveSetUp aufrufende Agententhread nicht, wie oben beschrieben und in MOLE realisiert, blockiert sondern terminiert, um den Agenten evtl. auslagern zu können. Kann eine Session etabliert werden oder tritt ein Timeout auf, so wird der Agent wieder eingelagert. Dem Agenten wird dann durch Aufruf einer festgelegten Methode der Aufbau der Session mitgeteilt bzw. eine Fehlernachricht über den aufgetretenen Timeout gesandt.

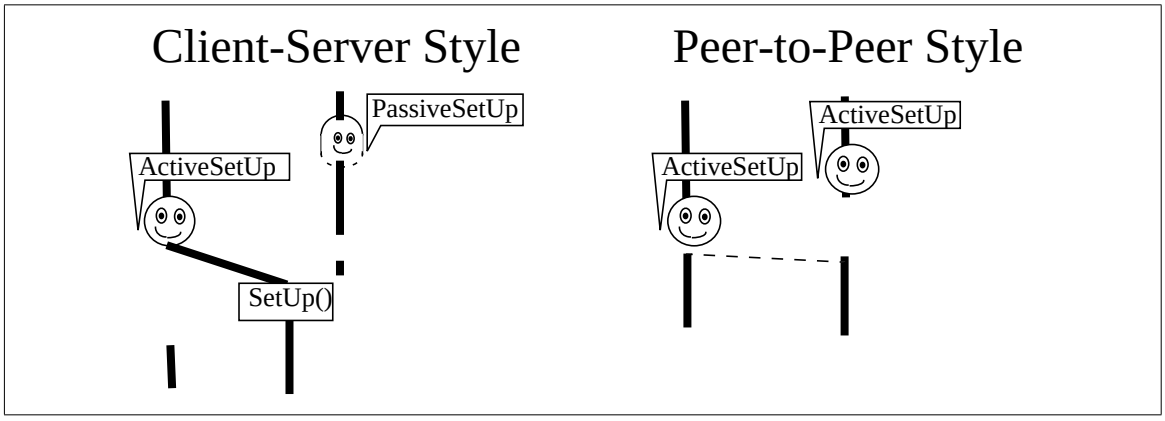


Abbildung 4-4: Verschiedene Interaktionsmuster durch PassiveSetUp und ActiveSetUp [BHRRSS]

Interaktion der beiden beteiligten Agenten

Nachdem eine Session etabliert wurde können die beteiligten Agenten im Rahmen der Session durch entfernten Methodenaufruf und Nachrichtenaustausch miteinander kommunizieren, bis die Session terminiert wird. Da Aufenthaltsort und Identität des Partners schon beim Aufbau geklärt wurden, vereinfacht sich diese Kommunikation etwas, da Ort und Identität des Partners nicht mehr angegeben werden müssen.

Abbau einer Session

Eine Session kann jederzeit durch jeden der beteiligten Agenten auch ohne eine vorherige Übereinkunft mit dem Partneragenten beendet werden. Dies sollte explizit durch Aufruf einer dafür vom Ort bereitgestellten Methode geschehen. Bei der Migration eines Agenten werden aber die Sessions, an denen dieser noch beteiligt ist, auch implizit terminiert. Die beteiligten Agenten werden in beiden Fällen durch Aufruf einer dafür vorgesehenen Methode über die Terminierung der Session informiert. Diese Methode kann vom Agentenprogrammierer überschrieben werden, um darauf adäquat reagieren zu können.

4.3 Mobilitätskonzepte

Neben der Kommunikation stellt die Mobilität der Agenten die zweite Säule eines mobilen Agentensystems dar. In diesem Abschnitt sollen die gängigen Mobilitätsformen vorgestellt werden. Die Darstellung stützt sich auf [BaHoSt] und [BHRS].

4.3.1 Remote-Execution und Code-on-Demand

Einen recht einfachen Mechanismus stellt die Remote-Execution dar. Bei ihr wird der Agent vor seiner Aktivierung auf den Zielrechner übertragen, wo er dann bis zu seinem Ableben verbleibt. Der Agent wird also nur einmal übertragen, wobei neben dem Programmcode des Agenten auch einige Parameter transferiert werden. Während seiner Ausführung kann der Agent mit Hilfe des Remote-Execution Mechanismus weitere Agenten starten. Der die Remote-Execution initiierende Agent bestimmt dabei das Ziel, auf dem das zu übertragene Programm ausgeführt wird. Im Gegensatz dazu initiiert beim Code-on-Demand das Ziel selbst das Herunterladen eines Programms von einem Server, der dieses anbietet. Die wohl bekanntesten Vertreter des Code-on-Demand Prinzips sind Java-Applets und ActiveX-Komponenten. Sowohl Remote-Execution als auch Code-on-Demand unterstützen zwar die Übertragung von Programmcode, sind aber für den Transfer von Agenten ungeeignet.

4.3.2 Migration

Mit der Migration steht dem Agentensystem hingegen ein geeigneter Ansatz zur Verfügung, der es einem Agenten ermöglicht, seine Ausführung an einem anderen Ort fortzusetzen. Die Entscheidung zur Migration wird dabei vom Agenten selbst, und nicht vom Agentensystem, z.B. zum Zwecke der Lastverteilung, getroffen. Initiiert der Agent die Migration, so wird seine Ausführung vom gegenwärtigen Ort angehalten. Dann wird er zum angegebenen Zielort transportiert, wo seine Ausführung schließlich fortgesetzt wird. Man unterscheidet zwei Migrationsformen, die im folgenden erläutert werden.

“Starke” Migration

Den höchsten Grad an Mobilität stellt die “starke” Migration zur Verfügung. Bei ihr wird der komplette Agentenzustand, der sich aus den Agentendaten und dem Ausführungszustand des Agenten zusammensetzt, mitsamt dem Agentencode zum Zielort übertragen. Dort wird der Ausführungszustand restauriert, und der Agent kann seine Arbeit an der Stelle fortsetzen, die der Anweisung zur Migration folgt. Die Implementation dieser Migrationsform erfordert aber das Extrahieren und Wiedereinspielen des Ausführungszustandes. Dies wird von den meisten Interpretern, so auch von Suns JVM, nicht unterstützt. Man könnte zwar versuchen den Interpreter dahingehend zu

modifizieren. Voraussetzung dafür ist aber, daß der Ausführungszustand des Agenten vollständig im Datenzustand des Interpreters, und nicht zum Teil in dessen eigenem Ausführungszustand, repräsentiert wird. Ist dies nicht der Fall, z.B. wenn der Interpreter auf einem Recursive-Descent-Parser basiert, so ist die Ermittlung des Ausführungszustand so gut wie unmöglich. Selbst wenn der Interpreter modifiziert werden kann, müßte man diesen dann mit dem Agentensystem ausliefern und installieren. Ein weiterer Nachteil besteht darin, daß der Ausführungszustand insbesondere von Agenten, die mehrere Threads ablaufen lassen, recht umfangreich sein kann. Dadurch kann die "starke" Migration zu einer relativ kostspieligen Operation werden, und eines der Ziele von mobilen Agentensystemen, nämlich die Reduzierung von Kommunikationskosten, kann unterlaufen werden.

"Schwache" Migration

Die aufgeführten Probleme führten zur Entwicklung eines weniger mächtigen Migrationskonzeptes, der sogenannten "schwachen" Migration. Statt des kompletten Agentenzustands werden nur Daten und Programmcode des Agenten, aber nicht dessen Ausführungszustand, übertragen. Dadurch wird einerseits der Umfang der zu transportierenden Daten limitiert, andererseits entfällt die Extraktion des Ausführungszustands und die damit verbundenen Modifikation des Interpreters. Als Nachteil steht dem der vom Agentensystem auf den Agentenprogrammierer verlagerte Aufwand bei einer Migration gegenüber. Bei einer Migration müssen nämlich die dafür relevanten Teile des Ausführungszustands vom Agentenprogrammierer in den Agentendaten codiert werden, um am neuen Ort einen adäquaten Ausführungszustand konstruieren zu können. Führt ein Agent die Migrationsanweisung aus, so wird zunächst dessen agentenspezifische stop()-Methode aufgerufen. In dieser kann der Agent letzte Aktionen in Erwartung der Migration durchführen. Anschließend wird der Agent vom aktuellen Aufenthaltsort angehalten und an den angegebenen Zielort übertragen. Dort wird seine Ausführung eingeleitet und die start()-Methode des Agenten aufgerufen. Es bleibt dem Agentenprogrammierer überlassen, in dieser Methode mit Hilfe der Agentendaten den alten Agentenzustand zu rekonstruieren.

Auf Grund der Schwierigkeiten, welche die "starke" Migration bereitet, verwendet das neue Agentensystem, wie auch schon MOLE, die "schwache" Migration. In zukünftigen Arbeiten muß aber, gerade auch in Anbetracht der Möglichkeiten die das Extrahieren und Wiedereinspielen des Ausführungszustands eines Agenten zusätzlich bietet, geprüft werden, ob nicht doch auf die "starke" Migration umgestellt wird. Im Ausblick wird auf die sich ergebenden Möglichkeiten näher eingegangen.

Kapitel 5

Implementation des Agentensystems

Nachdem im vorangegangenen Kapitel die Konzeption des Agentensystems vorgestellt wurde, widmen sich folgende Abschnitte dessen Implementation. Dieses Kapitel ist dabei ähnlich gegliedert wie das vorangegangene.

Das Agentensystem ist in einigen Punkten nicht nur konzeptionell eng an MOLE angelehnt. Bei dessen Implementation wurde auch versucht, möglichst viel Programmcode aus dem MOLE-Projekt wiederzuverwenden. Gerade in den Bereichen, die durch die veränderte Architektur des Agentensystems wenig oder gar nicht betroffen waren, wie z.B. Agentennamen oder Badges, ist dies sehr gut gelungen. Aber auch in den anderen Bereichen, wie z.B. bei der Location-Klasse, war dies möglich, wenngleich hier jedoch größere Änderungen durchgeführt werden mußten. Die nun folgende Beschreibung des Agentensystems erhebt keinen Anspruch auf Vollständigkeit, da dies den Rahmen dieser Arbeit sprengen würde. Es werden aber die interessanten Aspekte des Systems vorgestellt, wozu gerade auch die Neuerungen und Änderungen gegenüber MOLE gehören.

5.1 Komponenten des Agentensystems

Begonnen wird die Systembeschreibung mit den Komponenten, aus denen sich das Agentensystem zusammensetzt. In weiteren Abschnitten wird dann auf die Implementation von Kommunikation und Migration eingegangen.

5.1.1 Die Message-Queue

Zum sicheren Verwahren der Agenten kommt eine Tuxedo-Message-Queue zum Einsatz. Diese wird in der Tuxedo Konfigurationsdatei spezifiziert. In Tuxedo werden Message-Queues in sogenannten Queue-Spaces verwaltet. In der Konfigurationsdatei werden die Namen der Message-Queue und des verwendeten Queue-Spaces angegeben. Für letztgenannten werden zusätzlich die Höchstzahlen der zu verwaltenden Queues und der abzuspeichernden Nachrichten spezifiziert. Außerdem können einige Einstellungen des Tuxedo-Servers, der die Message-Queue verwaltet (nicht zu verwechseln mit dem Q-Manager), vorgegeben werden.

5.1.2 Die Serverklasse

Auch die Serverklasse wird in der Tuxedo Konfigurationsdatei konfiguriert. Hier werden unter anderem die Anzahl der zu startenden Serverprozesse, das Programm, das von diesen ausführt werden soll, und die Parameter, die diesem Programm übergeben werden sollen, angegeben. Das Serverprogramm wurde mit dem buildserver-Kommando aus dem Code für den ExecuteAgent-Dienst und den benötigten Bibliotheken (Tuxedo und Java) erzeugt und erwartet als Parameter den Dateinamen einer Konfigurationsdatei für die Sublocation.

Wie in Kapitel zwei schon erwähnt, sieht Tuxedo bei der Implementation eines Servers zur Zeit nur den Einsatz der Programmiersprachen C, C++ und COBOL, nicht aber den von Java, vor. Deshalb wurde das Grundgerüst des Servers in C realisiert. In der tpsvrinit()-Funktion, die nur einmal beim Start des Serverprozesses vom Tuxedo-System aufgerufen wird, wird durch Aufruf der Java-Native-Interface (JNI)-Funktionen GetDefaultJavaVMInitArgs() und CreateJavaVM eine Java-Virtual-Machine gestartet. Außerdem wird der Name der übergebenen Konfigurationsdatei für den späteren Zugriff in einer Variablen gespeichert. Die JVM wird in der tpsvrclose()-Funktion, die von Tuxedo beim Beenden des Serverprozesses aufgerufen wird, durch die JNI-Funktion DestroyJavaVM wieder gelöscht, kann aber bis zu diesem Zeitpunkt vom ExecuteAgent-Dienst zur Ausführung von Java-Klassen benutzt werden. Der ExecuteAgent-Dienst erwartet als Parameter einen typisierten Puffer des vordefinierten Tuxedo-Typs CARRAY. Wie der Name schon andeutet, ist dies ein Feld dessen Elemente vom Typ Character sind. Dieses enthält den serialisierten Agenten, der ausgeführt werden soll. Durch die JNI-Funktionen FindClass(), GetStaticMethodID() und CallStaticVoidMethod() wird eine Sublocation gestartet, der dieses Feld und der oben erwähnte Dateiname übergeben werden. Wird die Sublocation beendet, so gibt sie ein Byte-Feld zurück. Dessen erster Wert zeigt an wie weiter zu verfahren ist. Geriet der Agent in einen Wartezustand, so enthält der Rest des Feldes den serialisierten Agenten, der dann mit tpreturn() an den Q-Manager zurückgegeben und von diesem in die Message-Queue eingetragen wird. Beendete sich der Agent oder migrierte er, so wird dies dem Q-Manager durch einen entsprechenden Rückgabewert mittels tpreturn() signalisiert.

5.1.3 Der Q-Manager

Die Funktionalität des Q-Manager wird durch einen Tuxedo Server erbracht. In der Tuxedo Konfigurationsdatei wird das Serverprogramm und als Parameter die Anzahl der zur Serverklasse gehörenden Serverprozesse spezifiziert. Da für die Vermittlung zwischen Message-Queue und Serverklasse praktisch nur ATMI-Funktionen verwendet werden, ist das Serverprogramm komplett in C geschrieben.

In einer Schleife wird wie folgt vorgegangen: Falls die Anzahl der bereits gestarteten Agenten kleiner als die Zahl der Serverprozesse ist, wird eine neue Transaktion gestartet und versucht, aus der Message-Queue den nächsten mit dem Correlation Identifier “zur Ausführung bereit” versehenen Agenten auszulesen. Schlägt dies fehl, weil kein solcher Agent existiert, so wird die Transaktion abgebrochen. Andernfalls wird die Ausführung des ausgelesenen Agenten durch asynchronen Aufruf des ExecuteAgent-Dienstes der Serverklasse angestoßen. Dieser Aufruf liefert einen Deskriptor zurück, der in einem Feld abgelegt wird und mit dem später das Resultat dieses Dienstaufrufs abgefragt werden kann. Nach dem Aufruf wird die Transaktion mit tpsuspend() suspendiert, um bei einem erneuten Schleifendurchlauf weitere Agenten starten bzw. bereits durchgeführte Dienstaufrufe abschließen zu können. Der von tpsuspend() zurückgelieferte Transaktionsdeskriptor wird ebenfalls in einem Feld gespeichert. In einer Schleife wird mit Hilfe der erwähnten Deskriptoren und der ATMI-Funktion tpgetrply() auf das Ende der gestarteten, aber noch ausstehenden ExecuteAgent-Dienstaufrufe geprüft. Vor Aufruf von tpgetrply() wird aber mit Hilfe der ATMI-Funktion tpresume() und dem gespeicherten Transaktionsdeskriptor noch die zugehörige Transaktion wiederbelebt. Ist der Dienstaufruf noch nicht abgeschlossen wird die Transaktion wieder suspendiert. Andernfalls wird je nach Rückgabewert des Dienstes gegebenenfalls der zurückgelieferte Agent mit dem Correlation Identifier “blockiert” in die Message-Queue geschrieben. Anschließend wird die Transaktion mit tpcommit() abgeschlossen.

Abb. 5-1 zeigt das Vorgehen des Q-Managers an einem Beispiel. Durchgezogene Linien symbolisieren dabei laufende, Balken blockierte Prozesse. Der Q-Manager liest den nächsten Agenten (A1) aus der Message-Queue und stößt dessen Ausführung durch Aufruf des ExecuteAgent-Dienstes an. Der Agent wird von Serverprozeß 1 der Serverklasse ausgeführt. Mit tpgetrply wird geprüft, ob der Dienst bereits beendet ist. Dies ist aber noch nicht der Fall. Der Q-Manager leitet die Ausführung eines weiteren Agenten (A2) ein. Dieser wird von Serverprozeß 2 ausgeführt. Beim zweiten Aufruf von tpgetrply() ist die Ausführung von A1 bereits beendet, so daß ein Resultat vorliegt. Im Beispiel geriet A1 in einen Wartezustand und wird deshalb in serialisierter Form zurückgegeben und vom Q-Manager in die Message-Queue geschrieben. Danach wird auf das Ende der Ausführung von Agent A2 geprüft. A2 sei beispielsweise migriert, so daß der Q-Manager nur die Transaktion abschließt. Im der Abbildung ist die Umschaltung der Transaktionen gut zu erkennen.

Trat während der Ausführung des ExecuteAgent-Dienstes ein ernsthafter Fehler auf, oder wurde die beim Start der Transaktion angegebene Timeout-Zeit überschritten wird die Transaktion vom Tuxedo System mit dem Attribut

“abort-only” markiert. Tpcommit() liefert in einem solchen Fall einen Fehlercode zurück. Dieser wird abgefragt und die Transaktion wird mit tpabort() explizit abgebrochen.

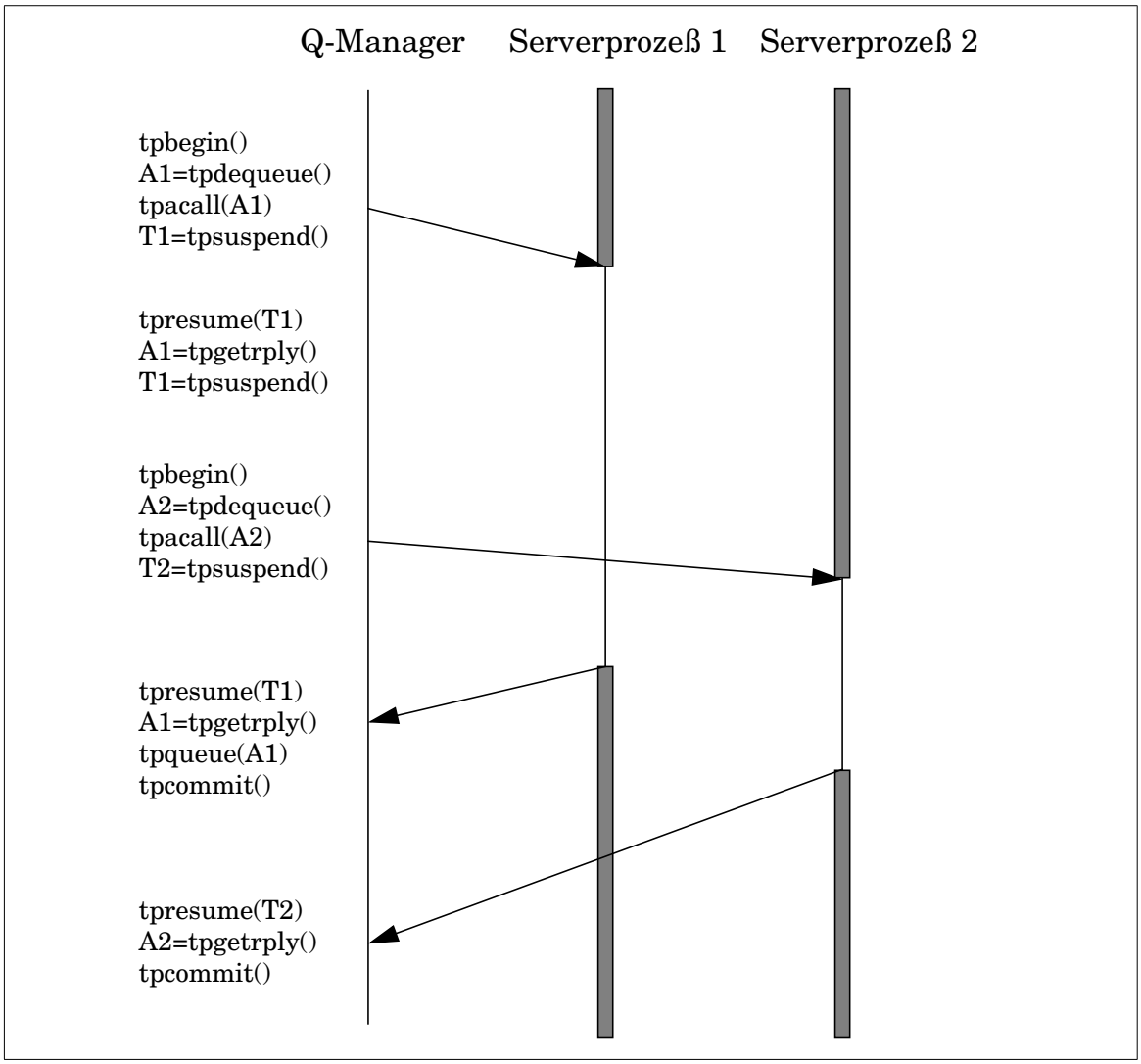


Abbildung 5-1: Die Arbeitsweise des Q-Mangers

5.1.4 Locations

Auch die Funktionalität eines Ortes wird durch einen Tuxedo Server realisiert. In der Tuxedo Konfigurationsdatei wird pro Ort ein Server eingetragen. Angegeben wird dort das Serverprogramm und als dessen Parameter der Name einer Datei, die Daten für die Konfiguration des Ortes, z.B. den Ortsnamen oder den Namen der zu verwendenden Message-Queue, enthält. Das Serverprogramm startet eine JVM und läßt durch diese die Java-Klasse Location ausführen, die gleich näher beschrieben wird.

Um die Ausführung von Agenten auch nach Absturz und anschließendem Wiederhochfahren eines Ortes sicherzustellen, müssen die vom Ort verwalteten Daten über die sich auf ihm aufhaltenden Agenten sicher gespeichert werden, um mit diesen Daten nach dem Wiederanfahren des Ortes weiterarbeiten zu können. Für das Speichern bietet sich eine Datenbank an. In der gegenwärtigen Implementation wird auf das sichere Ablegen der Daten allerdings verzichtet.

Starten eines Ortes

Beim Abarbeiten der main()-Methode werden die Konfigurationsdatei des Ortes eingelesen und Initialisierungen durchgeführt. So werden unter anderem die Namen des Ortes und der zu verwendenden Message-Queue gesetzt oder die Ausgabe von Debug-Meldungen eingeschaltet. Durch Erzeugen eines Objektes der Klasse ResolverFake, das Zeichenketten auf (IP-Adresse, Port)-Paare abbildet und mit einer Tabelle aus (Ortsname, IP-Adresse, Port)-Tupeln initialisiert wird, können Ortsnamen aufgelöst werden. Bevor die main()-Routine schließlich in den Eingabemodus übergeht, in dem der Benutzer unter anderem neue Agenten erzeugen oder auch die Location beenden kann, wird ein Objekt der Klasse Location erzeugt. Bei der Ausführung des Konstruktors werden weitere Initialisierungen vorgenommen. Dazu gehört die Anmeldung der Location unter dem Ortsnamen bei der lokalen RMI-Registry, um Sublocations und anderen Orten die Kontaktaufnahme mit ihr und den Zugriff auf die von ihr bereitgestellte Methoden zu ermöglichen.

Erzeugen eines Agenten

Eine dieser Methoden, die dem Erzeugen eines neuen Agenten dient, ist startNewAgent(). Sie kann sowohl von anderen Agenten über deren Sublocations aufgerufen werden, aber auch beim Erzeugen eines Agenten über die Kommandozeile des Ortes wird sie benutzt. Als Parameter erwartet sie den Namen der Agentenklasse und eine Datenstruktur, welche die Parameter für die Initialisierungsmethode des zu erzeugenden Agenten enthält. Sie versucht eine Instanz der angegebenen Klasse zu generieren und ruft die createAgent()-Methode auf, falls dies gelingt. In dieser wird ein neu erzeugter Agentenname an den Agenten vergeben und dessen Initialisierung durch Aufruf seiner init()-Methode angestoßen, wobei die Datenstruktur mit den Parametern übergeben wird. Außerdem wird der Anmeldungszähler des Agenten, welcher der Erkennung von Agentenabstürzen dient und später beschrieben wird, auf Null gesetzt. Trat während createAgent() kein Fehler auf, so wird schließlich die _arrive()-Methode ausgeführt, die auch bei der Ankunft eines Agenten als Folge einer Migration zum Einsatz kommt. Um die Sublocation, die den Agenten später ausführt, über dessen Aufenthaltsort zu informieren, speichert _arrive() Name und RMI-URL des Ortes in den Daten des Agenten. Mit Hilfe des RMI-URL kann die Sublocation später mit dem Ort kommunizieren. Danach wird die addStartingAgent()-Methode einer Instanz der Klasse AgentGuestlist auf-

gerufen. Diese Instanz wurde vom Konstruktor der Location erzeugt. Sie dient der Verwaltung der sich auf dem Ort aufhaltenden Agenten, wozu insbesondere das Wiedereinlagern von Agenten, die sich in der Message-Queue befinden und für die beispielsweise ein Kommunikationswunsch vorliegt, gehört. Die addStartingAgent()-Methode trägt einige Informationen über den Agenten, wie z.B. dessen Agentennamen, den Stand des Anmeldungszählers und die Information, daß beim Ausführen des Agenten dessen start()-Methode aufzurufen ist, in eine Hashtable ein, wobei der Agentenname als Schlüssel fungiert, und schreibt den Agenten mit dem Correlation Identifier “zur Ausführung bereit” in die Message-Queue.

An- und Abmelden eines Agenten durch dessen Sublocation

Damit der Ort über den aktuellen Zustand der sich auf ihm aufhaltenden Agenten informiert ist, melden die Sublocations bei ihrem Start den Agenten, den sie ausführen, bei dessen Aufenthaltsort an. Der Ort stellt hierzu die registerAgent()-Methode zur Verfügung. Als Parameter werden der Agentenname, der RMI-URL, unter dem die Sublocation des Agenten erreichbar ist, und der Anmeldungszähler des Agenten übergeben. RegisterAgent() ruft die changeAgentStateStartingToActive()-Methode der AgentGuestlist Klasse auf. Diese aktualisiert die den Agenten betreffenden Daten (der RMI-URL wird gespeichert und der Agentenzustand auf “Aktiv” gesetzt) und unterbricht auf den Agenten wartende Threads durch Aufruf der interrupt()-Methode der Threadklasse. Anschließend vergleicht registerAgent() den übergebenen mit dem vom Ort gespeicherten Anmeldungszähler des Agenten. Stimmen beide überein, so erhöht die registerAgent()-Methode den Wert des vom Ort verwalteten Zählers um eins. Nach Rückkehr aus dieser Methode wird der Anmeldungszähler des Agenten durch die Sublocation ebenfalls um eins erhöht und die durch den von registerAgent() zurückgegebenen Wert spezifizierte Aktion durchgeführt. Auf diese Aktionen wird im Abschnitt über Sublocations näher eingegangen. Stimmen die beiden Anmeldungszähler nicht überein, so muß der Agent nach seiner letzten Anmeldung abgestürzt sein. Durch einen entsprechenden Rückgabewert wird die Sublocation veranlaßt, die agentRecovery()-Methode des Agenten aufzurufen. Außerdem wird der Anmeldungszähler des Agenten auf den Stand des vom Ort verwalteten Zählers gebracht.

Sublocations melden Agenten vor deren Ausführung nicht nur an. Wenn sie die Ausführung des Agenten beenden, melden sie ihn bei dessen Aufenthaltsort auch wieder ab. Dazu bietet der Ort die unregisterAgent()-Methode an. Ihr wird neben dem Agentennamen auch ein Integerwert übergeben, der den Ort über den aktuellen Zustand des Agenten informiert. Folgende Werte sind möglich: der Agent hat sich beendet, der Agent ist auf einen anderen Ort migriert oder der Agent geriet in einen Wartezustand und wird daher in die Message-Queue eingetragen. In den beiden zuerst genannten Fälle wird die removeAgent() der AgentGuestlist Klasse aufgerufen. Da sich der Agent zu diesem Zeitpunkt nicht in der Message-Queue befindet und auch danach nicht wieder

hineingeschrieben wird, löscht diese Methode nur die gespeicherten Informationen des angegebenen Agenten aus den Datenstrukturen des Ortes. Geriet der Agent in einen Wartezustand, so wird die changeAgentStateActiveToQueued()_Methode der AgentGuestlist Klasse aufgerufen, welche die Informationen über den Agenten auf den neusten Stand bringt.

Vermittlung eines Kommunikationspartners

Wie im vorangegangenen Kapitel bereits erläutert, muß ein Agent zumindest bei der ersten Kontaktaufnahme mit einem Kommunikationspartner den Aufenthaltsort des Partneragenten kontaktieren. Wird der Partner durch eine Badge spezifiziert oder ist die gecachte Sublocationreferenz veraltet, so muß auch sonst der Ort des Kommunikationspartners in die Kommunikation mit einbezogen werden. Der Ort stellt hierfür die getAgentRMIURL()-Methode zur Verfügung, die den RMI-URL, unter dem die Sublocation des übergebenen Agenten erreichbar ist, zurückgibt. Der Agent wird gegebenenfalls vorher zur Ausführung gebracht. Als Parameter erwartet sie den Ortsnamen des Partneragenten, einen Agentennamen bzw. eine Badge zur Spezifikation des Kommunikationspartners und eine Timeout-Zeit.

Stimmt der angegebene Partnerort mit dem aktuellen nicht überein, so wird mit Hilfe der ResolverFake Klasse und der RMI-Registry die getAgentRMIURL()-Methode des Zielortes aufgerufen. Falls der Kommunikationspartner durch eine Badge spezifiziert wurde, wird vom Ort aus den Agenten, die diese Badge tragen, einer ausgewählt. Existiert kein solcher Agent, so wird eine NoSuchAgentException geworfen. Falls ein Agent ausgewählt werden konnte, oder der Partner durch einen Agentennamen spezifiziert wurde, wird die getAgentRMIURL()-Methode der AgentGuestlist Klasse aufgerufen, die wie folgt abläuft: Der Agent wird in der oben erwähnten Hashtable, die Informationen über die sich auf dem Ort aufhaltenden Agenten enthält, gesucht. Falls die Suche erfolglos ist, wird eine NoSuchAgentException geworfen. Andernfalls werden die über den Agenten angelegten Informationen analysiert. Wird der Agent bereits ausgeführt, so gibt getAgentRMIURL() den in den Informationen enthaltenen RMI-URL zurück. Befindet sich der Agent in der Message-Queue, wurde seine Aktivierung aber bisher noch nicht angestoßen, so wird dies nun getan, indem der Agent im Rahmen einer Transaktion aus der Message-Queue gelesen und sofort wieder, diesmal aber mit dem Correlation Identifier “zur Ausführung bereit”, hineingeschrieben wird. Danach wird, wie auch für den Fall, daß der Agent sich noch in der Message-Queue befindet, seine Ausführung aber bereits angestoßen wurde, auf die Anmeldung des Agenten am Ort durch seine Sublocation gewartet. Dazu wird der ausführende Thread in eine Hashtable eingetragen, deren Schlüssel Agentennamen und deren Elemente Vektoren mit den auf sie wartenden Threads sind. Anschließend wird der Thread für die getAgentRMIURL() übergebene Zeitspanne schlafen gelegt. Meldet sich in der Zwischenzeit der Agent an, so wird der Thread durch eine InterruptedException geweckt. Nun liegt der RMI-URL des Agenten vor und wird zurückgege-

ben. Verstreicht die Timeout-Zeitspanne, ohne daß sich der Agent am Ort anmeldet, so wird der Thread aus der Hashtable entfernt, und es wird eine TimeoutException geworfen.

Die Sleep-Funktion

Müssen Agententhreads für eine gewisse Zeit warten, so sollten sie dazu nicht die sleep()-Funktion der Thread-Klasse benutzen, sondern die des Agentensystems. Dazu stellt die Sublocation eine sleep()-Methode zur Verfügung, die als Parameter eine in Millisekunden angegebene Zeitspanne erwartet. Die Sublocation leitet den Aufruf an den Aufenthaltsort des Agenten weiter, wobei der Name des aufrufenden Agenten mit übergeben wird. Der Ort erzeugt und startet einen neuen Thread und kehrt zurück. Die Sublocation beendet dann den aufrufenden Agententhread, so daß der Agent ausgelagert werden kann. Der auf dem Ort ablaufende neue Thread legt sich für die angegebene Zeitspanne schlafen. Ist diese verstrichen, so wird durch Aufruf von getAgentRMIURL() der Agent falls nötig wieder eingelagert. Anschließend wird die wakeUp()-Methode der den Agenten ausführenden Sublocation aufgerufen. Diese startet einen neuen Thread, in dem die wakeUp()-Methode des Agenten aufgerufen wird.

Weitere vom Ort angebotene Methoden

Der Ort stellt Agenten bzw. den diese ausführenden Sublocations weitere Methoden zur Verfügung. So spielen bei der Kommunikation und Migration vom Ort angebotene Dienste eine wichtige Rolle. Auf diese wird aber in eigenen Abschnitten am Ende dieses Kapitels näher eingegangen.

5.1.5 Sublocations

Die Sublocation ist die Systemkomponente, die den Agenten ausführt und diesem die Dienste des Ortes zugänglich macht. Aus Sicht des Agenten verhält sich die Sublocation wie sein Aufenthaltsort. Um diese Transparenz zu erreichen, stellt die Sublocation dem Agenten für jede vom Ort angebotene Methode ein Pendant zur Verfügung, das den Aufruf an den Aufenthaltsort des Agenten weiterreicht.

Starten einer Sublocation

Wie im Abschnitt über die Serverklasse beschrieben, wird eine Sublocation durch den ExecuteAgent-Dienst der Serverklasse gestartet. Dabei wird ihr der auszuführende Agent in einem Byte-Feld und der Name einer Konfigurationsdatei übergeben. Diese Datei wird gelesen und entsprechende Initialisierungen vorgenommen. So wird beispielsweise der Name der zu verwendenden Message-Queue gesetzt und , wie auch beim Starten eines Ortes, eine Instanz der ResolverFake Klasse erzeugt und initialisiert. Außerdem wird ein sogenanntes MCP (Master Control Programm) gestartet, eine Subklasse von Thread, die für

die Zuteilung von Rechenzeit an die einzelnen Agententhreads zuständig ist. Danach wird aus dem Byte-Feld ein Objekt der Klasse Agent generiert. Auch an diesem werden Initialisierungen vorgenommen. Die Sublocation meldet sich dann unter dem Namen des Agenten bei der lokalen RMI-Registry an, um für dessen Aufenthaltsort und die Kommunikation mit anderen Agenten erreichbar zu sein. Im Agenten ist der RMI-URL seines Aufenthaltsortes gespeichert. Die Sublocation ermittelt damit bei der RMI-Registry des Ortes eine Referenz auf diesen, mit dem dann durch Aufruf von registerAgent() der Agent bei seinem Aufenthaltsort angemeldet wird. Der Rückgabewert der registerAgent()-Methode gibt an, wie die Sublocation weiter zu verfahren hat. Beim erstmaligen Ausführen des Agenten nach seiner Erzeugung oder einer Migration ist dies der Aufruf seiner start()-Methode. Stellte der Ort eine Diskrepanz zwischen seinem und dem Anmeldungszähler des Agenten fest, so wird die agentRecovery()-Methode des Agenten aufgerufen. Falls der Agent wegen eines Kommunikationswunsches oder auf Grund des Verstreichens der bei einem Sleep-Aufruf angegebenen Zeitspanne wieder eingelagert wird, so tut die Sublocation nichts, da der Kommunikationspartner bzw. der Aufenthaltsort die Initiative ergreift und beispielsweise die von der Sublocation bereitgestellte Methode zur Nachrichtenauslieferung oder die wakeUp()-Funktion aufruft.

Beenden einer Sublocation

Nicht nur beim Beenden des Agenten oder bei dessen Migration auf einen anderen Ort sollte sich die den Agenten ausführende Sublocation beenden, sondern auch wenn der Agent in einen Wartezustand gerät. Wie im vorangegangen Kapitel beschrieben, setzt das Agentensystem dabei auf die Kooperation des Agenten bzw. dessen Programmierer. Dieser sollte wartende Agententhreads entweder explizit oder implizit, z.B. durch die Sleep-Funktion, beenden. Stellt das von der Sublocation gestartete MCP, das für die Verwaltung der Agententhreads verantwortlich ist, fest, daß kein Agententhread mehr existiert, so ruft es die shutdownSublocation()-Methode der Sublocation auf, um diese zu beenden. Da bei der Migration oder beim Beenden eines Agenten alle Agententhreads terminiert werden, kann das Beenden der Sublocation einheitlich über diese Methode abgewickelt werden. Um die verschiedenen Situationen zu unterscheiden, wird beim Beenden oder Migrieren des Agenten eine Variable gesetzt. ShutdownSublocation() meldet den Agenten bei dessen Aufenthaltsort durch Aufruf der unregisterAgent()-Methode des Ortes ab. Neben dem Agentennamen wird ihr der Zustand des Agenten, der aus der eben erwähnten Variablen ermittelt werden kann, mitgegeben. Anschließend meldet sich die Sublocation bei der lokalen RMI-Registry ab. Geriet der Agent in einen Wartezustand, so wird er in das beim Start übergebene Byte-Feld geschrieben. Die Sublocation wird durch Rückgabe dieses Feldes beendet.

Die Terminierung der Sublocation wird eingeleitet, sobald das MCP feststellt, daß keine Agententhreads mehr registriert sind. Dies hätte aber zur Folge, daß direkt nach dem Start des MCP, also kurz nach Start der Sublocation, diese

auch gleich wieder beendet würde, da zu diesem Zeitpunkt noch keine Agententhreads vorhanden sind. Deshalb wird nach dem Erzeugen des MCP-Objektes, aber noch vor dessen Start bei diesem ein sogenannter RespiteThread registriert, der sich für eine festgelegte Zeit schlafen legt und dann terminiert. Dadurch wird die Sublocation frühestens nach dieser Zeit beendet. Ein weiterer Vorteil dieser Lösung ist, daß nach einem Wartezustand wieder eingelagerte Agenten ihren Serverprozeß nicht für unbestimmte Zeit blockieren können. Wie weiter oben beschrieben, warten diese auf Aktivitäten des Kommunikationspartners, der ihre Aktivierung einleitete. Falls sich dieser aber nicht meldet, weil er beispielsweise abgestürzt ist, wird durch den RespiteThread sichergestellt, daß der Agent nach der festgelegten Zeit wieder in die Message-Queue ausgelagert wird.

5.2 Kommunikation

In diesem Abschnitt sollen nun die dem Agenten zur Verfügung stehenden und im vorangegangenen Kapitel vorgestellten Kommunikationsmechanismen beschrieben werden. Im einzelnen sind dies der asynchrone und synchrone Nachrichtenaustausch, der entfernte Methodenaufruf und Sessions. Eine Gemeinsamkeit dieser Mechanismen besteht darin, daß diese ein umfangreiches Zusammenspiel der beteiligten Sublocations und des Aufenthaltsortes des Zielagenten erfordern. Bevor diese Mechanismen erläutert werden, soll aber auf die Realisierung von Agentennamen und Badges eingegangen werden.

5.2.1 Global eindeutige Agentennamen

Wie im vorangegangenen Kapitel erläutert, wird jeder Agent bei der Erzeugung mit einem Agentennamen versehen. Dieser ist unveränderlich und wird vom Agentensystem generiert. Agentennamen bestehen zur Zeit aus acht Bytes. Jedem Ort wird über dessen Konfigurationsdatei ein Bereich aus dem möglichen Namensraum zugewiesen, der vom Ort für die Namenvergabe genutzt werden kann. Um global eindeutige Agentennamen zu vergeben, muß sichergestellt sein, daß diese Bereiche disjunkt sind. Dafür sind zur Zeit die Administratoren der Orte des Agentensystems durch eine geeignete Konfiguration der Namensbereiche für jeden Ort zuständig.

5.2.2 Badges

Neben der Identifikation eines Kommunikationspartners durch dessen Agentennamen, ist es auch möglich, den Partner durch eine Badge zu spezifizieren. Badges werden durch einfache Zeichenketten realisiert. Mit den von der Sublocation angebotenen Methoden pinOn() und pinOff(), die jeweils eine Badge als Parameter erwarten, können sich Agenten Badges anheften bzw. diese wieder ablegen. Da Badges vom Aufenthaltsort des Agenten verwaltet werden, leitet

die Sublocation die Aufrufe an die registerBadge()- bzw. unregisterBadge()-Methode des Aufenthaltsortes weiter, wobei neben der Badge auch der Agentenname übergeben wird. RegisterBadge() trägt das (Badge, Agentenname)-Paar in einer Datenstruktur vom Typ BadgeTable ein, woraus es durch unregisterBadge() wieder gelöscht wird. Weitere vom Ort angebotene Methoden sind getBadges(), getBadgeVector() und providersOf(). Mit den zwei zuerst genannten Methoden kann der Agent eine Enumeration bzw. einen Vektor der am Ort registrierten Badges abfragen. ProvidersOf() liefert einen Vektor von Agentenamen der Agenten, welche die angegebene Badge zur Zeit tragen, zurück.

Wird bei einem Kommunikationswunsch der Partner durch eine Badge spezifiziert, so wählt der angegebene Partnerort aus den sich auf ihm befindenden und mit der Badge versehenen Agenten den nach einem Round-Robin-Verfahren nächsten Agenten aus.

5.2.3 Asynchroner Nachrichtenaustausch

Um eine Nachricht asynchron zu versenden stehen dem Agentenprogrammierer mehrere Möglichkeiten zur Verfügung. Die allgemeinste ist der Aufruf der von der Sublocation angebotenen message()-Methode. Als Parameter erwartet sie eine Timeout-Zeit und eine Nachricht der Klasse Message, welche die Namen von Sender- und Empfängeragent, deren Aufenthaltsort, die zu verwendende Fehlersemantik, eine Message-ID und das zu versendende serialisierbare Objekt enthält. Statt eines Agentennamens für den Empfänger kann auch eine Badge angegeben werden. Komfortabler gestaltet sich der asynchrone Versand mit den ebenfalls von der Sublocation angebotenen sendUnreliable()-, sendReliable()- und sendMailbox()-Methoden. Diese erwarten als Parameter Zielort, Agentenname des Empfängers bzw. eine Badge, eine Message-ID, das zu versendende Objekt und eine Timeout-Zeit. Mit diesen Daten, der durch den Methodennamen spezifizierten Fehlersemantik, Agentenname und Ort des Aufrufers wird eine neue Message erzeugt und intern die message()-Methode aufgerufen.

Die message()-Methode erzeugt einen neuen Thread der Klasse MessageThread, startet diesen und gibt den Kontrollfluß unmittelbar danach an den Aufrufer zurück. Der erzeugte Thread läuft dann parallel zur normalen Programmausführung ab. Er ruft die _synchmessage()-Methode der Sublocation auf, welche die Nachricht synchron versendet und im folgenden beschrieben wird.

Falls Sender- und Empfängeragent übereinstimmen, der Agent also eine Nachricht an sich selbst schickt, wird direkt die dispatchMessage()-Methode des Agenten aufgerufen, auf die weiter unten näher eingegangen wird. Ist die Nachricht an einen anderen Agenten gerichtet, so wird dieser im in Kapitel vier erwähnten Cache, der die Namen von Agenten auf deren letzte gültige Sublocationreferenz abbildet, gesucht. Ist diese Suche erfolgreich, so wird mit Hilfe die-

ser Referenz die deliverMessage()-Methode der Sublocation des Kommunikationspartners aufgerufen, welche die Nachricht an den Agenten ausliefert. Tritt bei diesem Aufruf eine RemoteException auf, so kommt das Verwenden der veralteten Referenz als Ursache dafür in Frage. Der Agent könnte sich beispielsweise wieder in der MessageQueue befinden. Für diesen und den Fall, daß der Cache keinen passenden Eintrag enthielt, muß die aktuelle Referenz des Empfängeragenten ermittelt werden. Dies ist auch vonnöten, wenn der Kommunikationspartner über eine Badge identifiziert wird. Hier ist die Verwendung des Caches unangebracht, da einerseits Badges auch wieder abgelegt werden können, und andererseits aus Lastverteilungsgründen nicht immer derselbe Agent ausgewählt werden sollte. Um die Referenz der Empfängersublocation zu ermitteln, muß der Aufenthaltsort des Kommunikationspartners kontaktiert werden. Befindet sich der Partneragent auf demselben Ort wie der Aufrufer, so ist eine Referenz dieses Ortes bekannt. Ist dies nicht der Fall, so wird mit Hilfe der Klasse ResolverFake der RMI-URL des Ortes ermittelt und durch Hinzuziehen der RMI-Registry eine Referenz auf den Ort abgefragt. Durch entfernten Methodenaufruf (RMI) von asynchmessage() des Aufenthaltsortes des Kommunikationspartners werden Nachricht und Timeout-Zeit übergeben. Asynchmessage() erzeugt und startet einen neuen Thread um den RMI-Aufruf zu beenden, da sonst eine mögliche Auslagerung des Senderagenten verhindert werden würde. In diesem Thread wird dann die getAgentRMIURL()-Methode des Ortes aufgerufen. Falls der Partneragent durch eine Badge angegeben wurde, wird ein passender Agent ausgewählt. Der Kommunikationspartner wird falls nötig zur Ausführung gebracht, dessen RMI-URL ermittelt und zurückgegeben. Anschließend wird die deliverMessage()-Methode der Empfängersublocation aufgerufen. Diese Methode erzeugt und startet einen neuen Thread und kehrt zurück, wodurch der von asynchmessage() gestartete Thread des Empfängerortes beendet wird. Der neue Thread ruft die dispatchMessage()-Method des Agenten auf.

Der Agent kann mehrere Empfangsmethoden für verschiedene Nachrichtenklassen, darunter auch benutzerdefinierte, zur Verfügung stellen. Diese Möglichkeit erleichtert die Arbeit des Agentenprogrammierers, da er z.B. die Behandlung von Fehlernachrichten durch die Bereitstellung einer entsprechenden Empfangsmethode von der übrigen Nachrichtenbearbeitung trennen kann. Die dispatchMessage()-Methode ist hierbei für den Aufruf der passenden, d.h. in Bezug auf die Nachrichtenklasse speziellsten, Empfangsmethode zuständig. Um die Menge der möglichen Nachrichtenklassen nicht einzuschränken, muß die Auswahl der Empfangsmethode dynamisch erfolgen. Zum Einsatz kommt dabei das Java-Reflection-API. Eine detailliertere Beschreibung ist in [Kubach] zu finden.

Während des gesamten hier dargestellten Ablaufs können an verschiedenen Stellen Fehler auftreten. Zum einen können bereits bei den von der Sendersublocation durchgeführten RMI-Aufrufen RemoteExceptions geworfen werden.

Andererseits können auch auf dem Empfangsort bei der Abarbeitung der asynchmessage()-Methode Exceptions entstehen. Dazu gehören RemoteExceptions bei RMI-Aufrufen, aber auch TimeoutExceptions beim Aufruf von getAgentRMIURL(). Je nach Entstehungsort werden sie entweder von der Sendersublocation oder dem Ort des Empfängers gemäß der angegebenen Fehlersemantik behandelt. Es wird also nichts weiter getan (Fehlersemantik Unreliable), die Nachricht in eine Fehlernachricht umgewandelt und mit der Fehlersemantik Unreliable an den Senderagenten geschickt (Reliable), oder die Nachricht wird durch Aufruf von storemessage() am Empfangsort gespeichert (Mailbox). Dort kann sie später vom Empfängeragenten durch Aufruf der vom Ort bereitgestellten deliverMessagesFor()-Methode abgeholt werden.

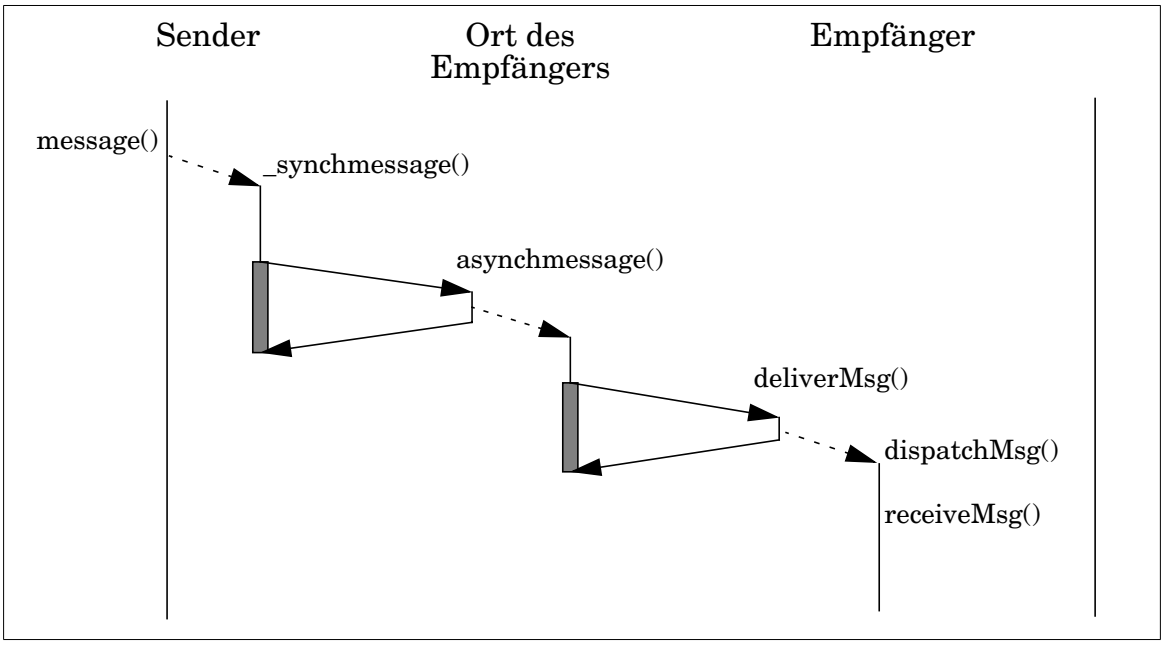


Abbildung 5-2: Asynchroner Nachrichtenversand über den Ort des Empfängers

Zur Verwendung des Caches ist noch anzumerken, daß dieser beim asynchronen Nachrichtenversand nicht aktualisiert werden kann, da RMI-URL und Referenz der Empfängersublocation vom Ort des Empfängers, und nicht von der Sublocation des Senders, die den Cache verwaltet, ermittelt werden. (Agentenname, Sublocationreferenz)-Paare werden nur beim synchronen Nachrichtenversand und entferntem Methodenaufruf in den Cache eingetragen. Nichtsdestotrotz können die bereits im Cache enthaltenen Referenzen auch beim asynchronen Nachrichtenaustausch verwendet werden.

Zwei Abbildungen sollen den Ablauf eines asynchronen Nachrichtenversands noch einmal veranschaulichen. Falls im Cache kein Eintrag für den Kommunikationspartner existiert, oder dieser durch eine Badge spezifiziert wurde, läuft der asynchrone Nachrichtenaustausch gemäß Abb. 5-2 ab. Senkrechte Striche verdeutlichen den Ablauf von Threads, Balken symbolisieren blockierte Threads, durchgezogene Pfeile dienen der Darstellung von RMI-Aufrufe und

gestrichelte Pfeile deuten schließlich das Starten neuer Threads an. Abb. 5-3 zeigt den Ablauf des asynchronen Nachrichtenversands für den Fall, daß der Kommunikationspartner durch einen Agentennamen spezifiziert wurde und sich im Cache der Sendersublocation ein Eintrag über den Partner befindet.

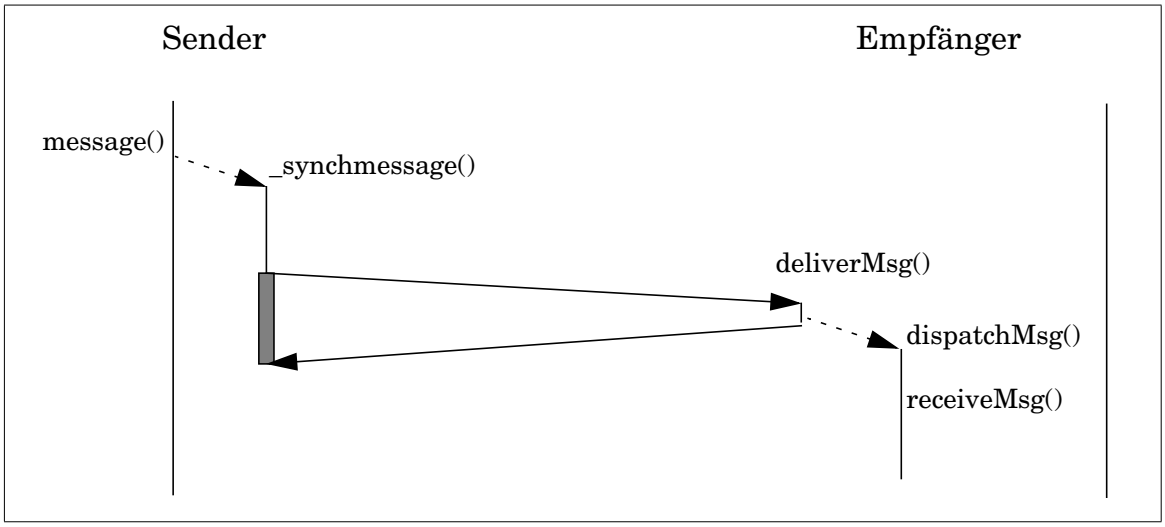


Abbildung 5-3: Asynchroner Nachrichtenaustausch bei Verwendung des Caches

5.2.4 Synchroner Nachrichtenaustausch

Für den synchronen Versand einer Nachricht stellt die Sublocation dem Agenten die synchmessage()-Methode zur Verfügung. Als Parameter erwartet sie, wie ihr asynchrones Pendant auch, eine Timeout-Zeit und eine Nachricht der Klasse Message, welche die Namen von Sender- und Empfängeragent, deren Aufenthaltsort, die zu verwendende Fehlersemantik, eine Message-ID und das zu versendende serialisierbare Objekt enthält. Statt eines Agentennamens für den Empfänger kann auch eine Badge angegeben werden. Im Gegensatz zum asynchronen Versenden wird kein neuer Thread erzeugt, sondern der Versand im Thread des Aufrufers durchgeführt, so daß dieser bis zur Rückkehr aus synchmessage() blockiert ist.

Ist die Nachricht an sich selbst gerichtet, wird direkt die dispatchMessage()-Methode des Agenten aufgerufen, welche wiederum die passende Empfangsmethode aufruft. Andernfalls wird, wie im vorangegangenen Abschnitt beschrieben, der Cache konsultiert. Enthielt dieser einen Eintrag für den Empfänger, so wird die deliverMessage()-Methode der Sublocation des Empfängers aufgerufen. Trat dabei eine RemoteException auf, schlug die Suche im Cache fehl oder wurde der Kommunikationspartner durch eine Badge angegeben, muß eine aktuelle Referenz der Empfängersublocation ermittelt werden. Dies geschieht durch Aufruf der getAgentRMIURL()-Methode des Empfängerortes, wobei falls nötig die Auswahl und Ausführung des Partneragenten angestoßen wird. Mit dem zurückgegebenen RMI-URL besorgt sich die Sendersublocation bei der

RMI-Registry eine Referenz der Empfängersublocation, die mit dem Agentennamen des Empfängers als Schlüssel in den Cache eingetragen wird. Anschließend wird die deliverMessage()-Methode der Sublocation des Empfängers aufgerufen, die wie früher erläutert die Auslieferung der Nachricht veranlaßt.

Während des beschriebenen Ablaufs können verschiedene Fehler auftreten, die durch Exceptions signalisiert werden. Sämtliche RMI-Aufrufe können eine RemoteException werfen, getAgentRMIURL() auch eine TimeoutException, falls der Partneragent nicht innerhalb der angegebenen Zeitspanne zur Ausführung gebracht werden kann, und eine NoSuchAgentException, falls der Partneragent sich nicht auf dem Empfängerort aufhält. Diese Exceptions werden an den Aufrufer von synchmessage(), d.h. den Agenten, weitergereicht und können von diesem abgefangen und entsprechend verarbeitet werden.

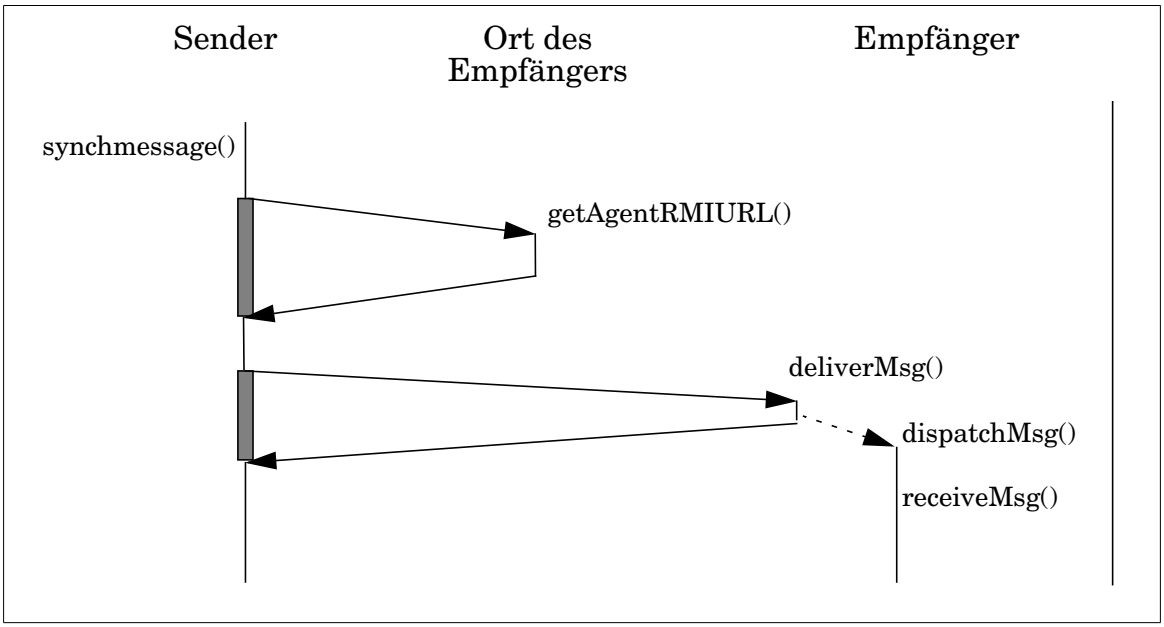


Abbildung 5-4: Synchroner Nachrichtenaustausch über den Ort des Empfängers

Abb. 5-4 zeigt den Ablauf eines synchronen Nachrichtenversands für den Fall, daß im Cache kein Eintrag für den Kommunikationspartner gefunden wird, oder dieser durch eine Badge spezifiziert wurde.

5.2.5 Entfernter Methodenaufruf

Für den entfernten Aufruf öffentlicher Methoden anderer Agenten stehen dem Agentenprogrammierer mehrere call()-Methoden der Sublocation zur Verfügung. Die allgemeinste dieser Methoden erwartet als Parameter den Namen der aufzurufenden Methode in Form einer Zeichenkette, einen Agentennamen bzw. eine Badge, um den Anbieter der entfernten Methode zu identifiziert, ein Feld mit den Parametern, mit der die entfernte Methode aufgerufen werden soll, und eine Timeout-Zeit. Um den Aufruf einer entfernten Methode für den

Agentenprogrammierer komfortabler zu machen, bietet die Sublocation weitere call()-Methoden, denen statt des Parameterfeldes null bis fünf Parameter der aufzurufenden Methode übergeben werden, so daß beim Aufruf von Methoden mit bis zu fünf Parametern das Anlegen und Füllen des Parameterfeldes entfällt. Diese call()-Methoden konstruieren das Parameterfeld, füllen es mit den übergebenen Parametern der entfernten Methode und rufen die zuerst genannte call()-Methode auf, die im folgenden beschrieben wird.

Falls der Zielagent durch einen Agentennamen spezifiziert wurde, konsultiert die Sublocation des Aufrufers ihren Cache. Findet sich darin der passende Eintrag, so wird die weiter unten beschriebene dispatchRPC()-Methode der Sublocation des Zielagenten aufgerufen. Trat dabei eine RemoteException auf, schlug die Suche im Cache fehl oder wurde der Zielagent durch eine Badge spezifiziert, muß eine aktuelle Referenz der Sublocation des Zielagenten ermittelt werden. Dazu wird die getAgentRMIURL()-Methode des Zielortes aufgerufen, die falls nötig die Auswahl und Ausführung des Zielagenten anstößt. Mit dem RMI-URL besorgt sich die Sublocation des aufrufenden Agenten eine Referenz der Sublocation des Zielagenten, die in den Cache eingetragen wird. Anschließend wird die dispatchRPC()-Methode der Sublocation des Zielagenten aufgerufen. Diese meldet den Thread beim MCP der Sublocation an, ruft die gewünschte Methode mit den angegebenen Parametern auf, meldet den Thread wieder ab und gibt das Resultat des Aufrufs zurück. Um die gewünschte Methode aufzurufen, wird das Java-Reflection-API eingesetzt. Detailliertere Information über den Ablauf der dispatchRPC()-Methode sind in [Kubach] zu finden.

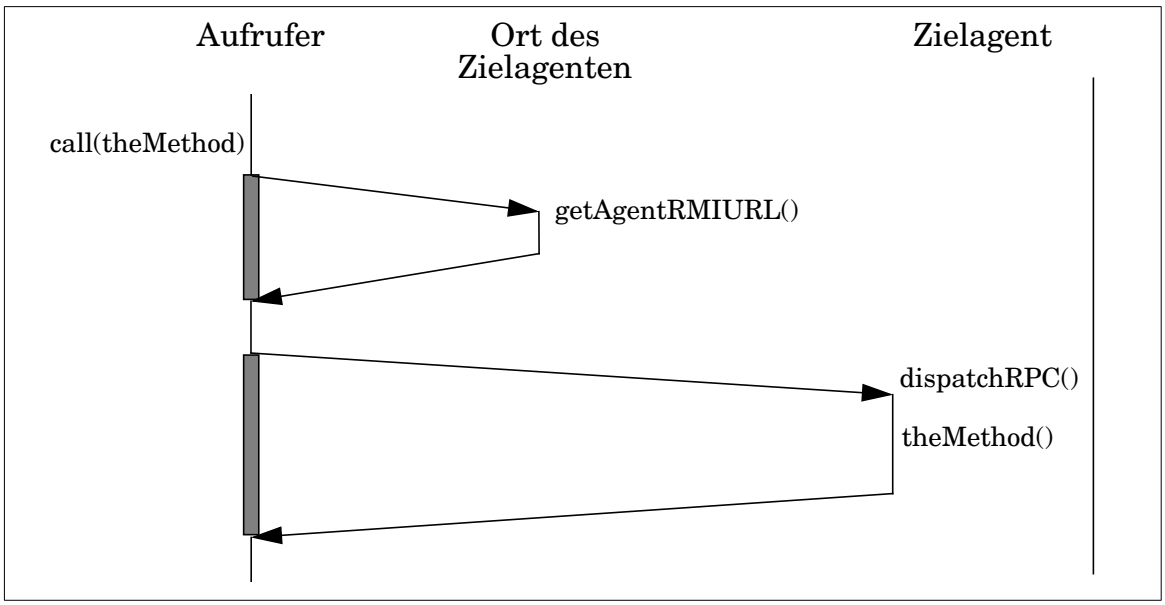


Abbildung 5-5: Entfernter Methodenaufruf über den Ort des Methodenanbieters

Wie beim synchronen Nachrichtenversand auch, werden auftretende Exceptions an den aufrufenden Agenten weitergereicht und können von diesem abgefangen und verarbeitet werden. Neben RemoteExceptions durch die verwendeten RMI-Aufrufe können auch NoSuchMethodExceptions, falls die aufzurufende Methode vom Agenten nicht angeboten oder nicht öffentlich ist, und NoSuchReceiverExceptions, falls sich der angegebene Zielagent nicht am Zielort aufhält bzw. kein Agent existiert, auf den die angegebene Badge paßt, entstehen.

Auch hier soll eine Abbildung der Veranschaulichung des beschriebenen Ablaufs eines entfernten Methodenaufrufs dienen. Abb. 5-5 tut dies für den Fall, daß im Cache kein Eintrag für den Kommunikationspartner existiert, oder dieser durch eine Badge spezifiziert wurde.

5.2.6 Sessions

Mit Sessions steht ein Mechanismus zur Verfügung, mit dem im Gegensatz zu den bisher besprochenen eine Kommunikationsbeziehung explizit aufgebaut wird.

Aufbau einer Session

Dazu bietet die Sublocation dem Agenten zwei Methoden an: passiveSetUp() und activeSetUp(). Als Parameter erwartet passiveSetUp() zur Spezifikation des Kommunikationspartners einen Agentennamen bzw. eine Badge und zusätzlich einen Integerwert, der die Höchstzahl der unmittelbar zu etablierenden Sessions festlegt. Der Aufruf wird von der Sublocation an den Aufenthaltsort des Agenten weitergeleitet, wobei zusätzlich dessen Agentenname übergeben wird. Der Ort prüft, ob durch den Aufruf von passiveSetUp() ausstehende Sessions etabliert werden können. Dazu durchsucht er eine von ihm verwaltete Datenstruktur, die ausstehende Sessions enthält, auf übereinstimmende Partnerspezifikationen. Werden passende Sessions gefunden, so werden diese etabliert, indem mit Hilfe eines getAgentRMIURL()-Aufrufs und der RMI-Registry eine Referenz der Sublocation des aufrufenden Agenten ermittelt und anschließend für jede zu etablierende Session die establishSession()-Methode der Sublocation aufgerufen wird. Diese startet einen neuen Thread, der seinerseits die sessionEstablished()-Methode des Agenten, die vom Agentenprogrammierer bereitgestellt wird, mit dem neu angelegten Session-Objekt als Parameter aufruft.

Während mit passiveSetUp() nur die Bereitschaft erklärt wird an einer Session teilzunehmen, können Agenten durch Aufruf von activeSetUp() Sessions initiieren. Die Sublocation bietet dem Agenten zwei Varianten von activeSetUp() an. Die erste erwartet als Parameter einen Agentennamen bzw. eine Badge, um den Kommunikationspartner zu spezifizieren, den Namen des Ortes, auf dem sich der Partneragent aufhält, und eine Timeout-Zeit. In der zweiten Variante

wird zusätzlich eine Badge angegeben, die es dem Agenten erlaubt sich bei der Auswahl des Sessionpartners durch diese Badge, und nicht durch seinen Agentennamen zu identifizieren. Beide Aufrufe werden von der Sublocation mit dem Namen des initiierenden Agenten versehen an den Aufenthaltsort des Agenten weitergeleitet. Der Ort erzeugt und startet einen neuen Thread und beendet den RMI-Aufruf der Sublocation, damit der aufrufende Agententhread terminiert werden kann, und eine Auslagerung des Agenten in die Message-Queue ermöglicht wird. Der neue Thread ruft makeSession() auf. Diese Methode prüft, ob eine ausstehende Session etabliert werden kann. Ist dies der Fall, so geschieht dies, wie bei passiveSetUp() beschrieben. Andernfalls wird die getSession()-Methode aufgerufen. Handelt es bei dem Partnerort um den Aufenthaltsort des aufrufenden Agenten, so wird der Aufruf lokal, ansonsten über RMI durchgeführt. Diese Methode ist blockierend und kehrt erst zurück, falls eine Session etabliert werden kann oder die Timeout-Zeitspanne verstrichen ist. Bei letzterem wird an den aufrufenden Agenten eine Fehlernachricht gesandt. Andernfalls wird die Session etabliert und der Agent durch Aufruf der establishSession()-Methode darüber informiert.

Interaktion der beiden beteiligten Agenten

Nachdem eine Session etabliert wurde können die beteiligten Agenten im Rahmen der Session durch entfernten Methodenaufruf und Nachrichtenaustausch miteinander kommunizieren bis die Session terminiert wird. Die Kommunikation wird über das erhaltene Session-Objekt abgewickelt, das message(), synchmessage() und call()-Methoden anbietet. Diese rufen intern ihre Pendants der Sublocation auf.

Abbau einer Session

Um eine Session zu beenden, stellt die Sublocation dem Agenten die terminateSession()-Methode zur Verfügung, der als Parameter die zu beendende Session übergeben wird. Die Sublocation reicht den Aufruf an die terminateSession()-Methode des Aufenthaltsortes des Agenten weiter. Diese ruft unregisterSession() auf, um den zweiten an der Session beteiligten Agenten über deren Termination zu informieren. Der Aufruf erfolgt je nach Aufenthaltsort des Partneragenten lokal oder über RMI. TerminateSession() ruft ihrerseits die sessionTerminated()-Methode der Sublocation des aufrufenden Agenten auf, die einen neuen Thread erzeugt und startet, der die sessionTerminated()-Methode des Agenten ausführt. Diese wird vom Agentenprogrammierer bereitgestellt. TerminateSession() entfernt schließlich die Session aus den Datenstrukturen des Ortes. Die unregisterSession()-Methode ruft analog die sessionTerminated()-Methode des zweiten an der Session beteiligten Agenten auf und löscht anschließend die Session aus den Datenstrukturen des Ortes. Das doppelte Löschen der Session durch terminateSession() und unregisterSession() liegt in der Verwaltung von Sessions durch zwei Sessionendpunkte begründet.

5.3 Migration

Wie im vorangegangenen Kapitel beschrieben, bietet das Agentensystem dem Agenten die Möglichkeit, durch “schwache” Migration seine Ausführung auf einem anderen Ort fortzusetzen. Dazu stellt die Sublocation dem Agenten eine goTo()-Methode zur Verfügung, die als einzigen Parameter den Namen des Zielortes entgegennimmt und im folgenden erläutert wird.

Um die Verwaltung von Sessions durch das Agentensystem zu vereinfachen, sollten an aktiven Sessions teilnehmende Agenten nicht migrieren. Tun sie dies doch so werden die beteiligten Sessions implizit beendet. Dazu wird die terminateSessionsOf()-Methode des derzeitigen Aufenthaltsortes des Agenten aufgerufen, wobei als einziger Parameter der Agentenname angegeben wird. Diese Methode durchsucht die SessionTable des Ortes und ruft für jede Session, an welcher der Agent beteiligt ist, die vorgestellte terminateSession()-Methode auf, die ihrerseits den beiden beteiligten Agenten die Beendigung der Session durch Aufruf deren sessionTerminated()-Methoden signalisiert. Nach Rückkehr aus terminateSessionsOf() wird die Ausführung des Agenten suspendiert, indem die suspendIt()-Methode des Agenten aufgerufen wird. Danach wird mit Hilfe der ResolverFake-Klasse der RMI-URL des Zielortes ermittelt. Mit diesem URL besorgt sich die Sublocation bei der RMI-Registry eine Referenz dieses Ortes. und ruft die handleMigration()-Methode des Zielortes auf, wobei der Agent als Parameter übergeben wird. HandleMigration() ruft die im Abschnitt Locations vorgestellte _arrive()-Methode auf. Diese setzt Name und RMI-URL des Ortes im Agenten, trägt Informationen über diesen, wie z.B. dessen Anmeldungszähler, in die Datenstrukturen des Ortes ein und speichert ihn mit dem Correlation Identifier “zur Ausführung bereit” in der Message-Queue ab. HandleMigration() gibt einen Wert vom Typ boolean zurück, der angibt ob die Aktionen erfolgreich durchgeführt werden konnten.

Trat beim oder während des Aufrufs von handleMigration() ein Fehler auf, so nimmt die Sublocation die Ausführung des Agenten durch Aufruf seiner resumeIt()-Methode wieder auf. Die Sessions, an denen der Agent beteiligt war, stehen aber nicht mehr zur Verfügung, da sie zu Beginn von goTo() beendet wurden. Andernfalls, also bei geglückter Migration, werden alle Agententhreads terminiert. Dies führt, wie im Abschnitt über Sublocations beschrieben, zum Beenden der Sublocation, wobei diese den Agenten bei seinem nunmehr alten Aufenthaltsort abmeldet.

Stürzt die Sublocation bzw. der sie ausführende Serverprozeß in der Zeitspanne zwischen erfolgreicher Rückkehr aus handleMigration() und dem schließen der beim ExecuteAgent-Dienstaufruf gestarteten Transaktion ab, so hat sich der Agent dupliziert. Er liegt in seinem aktuellen Zustand auf dem neuen Aufenthaltsort und nach dem Rollback der Transaktion in seinem alten in der Message-Queue vor. Diese Duplikation muß natürlich unterbunden werden. Eine Lösungsmöglichkeit, die allerdings noch nicht realisiert wurde, ist eine Rück-

meldung des neuen Aufenthaltsortes an den alten über die erfolgreiche Migration. Der alte Aufenthaltsort zieht dann falls nötig den abgestürzten Agenten aus dem Verkehr.

Bewertung

Kapitel 6

Nachdem in Kapitel vier die Konzeption und in Kapitel fünf die wichtigsten Aspekte der Implementation des neuen Agentensystems vorgestellt wurden, soll an dieser Stelle eine Bewertung erfolgen.

6.1 Zuverlässigkeit

Als gravierendster Nachteil von MOLE wurde dessen mangelnde Zuverlässigkeit erkannt. In MOLE besteht also die Möglichkeit, daß durch Prozeß- oder Systemabstürze Agenten verlorengehen können. Ursache hierfür ist die Tatsache, daß Agenten im Laufe ihrer Ausführung zu keinem Zeitpunkt in einem persistenten Zustand vorliegen. Im neuen Agentensystem werden Agenten nach dem Erzeugen, einer Migration oder wenn sie in einen Wartezustand geraten in einer Tuxedo Message-Queue gespeichert. Dadurch ist das Verlorengehen eines Agenten im neuen System nicht mehr möglich. Gerade im Hinblick auf einen Einsatz von Agentensystemen im kommerziellen Umfeld ist dieser Sachverhalt von zentraler Bedeutung, denn Agenten könnten beispielsweise auch elektronisches Geld mit sich führen.

Stürzt ein Agent ab, so wird die Transaktion, in deren Rahmen er ausgeführt wurde, abgebrochen, und der Agent liegt in dem zuletzt gespeicherten Zustand wieder in der Message-Queue vor. Der Agent wird später wieder zur Ausführung gebracht, wobei das Agentensystem feststellt, daß der Agent einen Absturz hinter sich hat. Dem Agenten bzw. dessen Programmierer wird durch Aufruf einer festgelegten Methode die Möglichkeit gegeben auf den Absturz adäquat zu reagieren. Der Nutzen dieses Ablaufs wird leider durch die Tatsache etwas geschmälert, daß bei komplexeren Aufgaben die Art und Weise, wie auf einen Absturz reagiert werden soll, für den Agenten nicht immer klar ersichtlich ist. Da der Agent zwar über den Absturz informiert wird, aber in der Regel keinen vollständigen Überblick über die bis zum Absturz durchgeführten Aktionen hat, kann die Fehlerbehandlung recht aufwendig sein. Dies kann dazu führen, daß der Agentenprogrammierer bewußt oder unbewußt einige Fehlersituationen vernachlässigt, was zu einer Vergeudung von Ressourcen des Agentensystems führen kann.

6.2 Skalierbarkeit

Neben der Zuverlässigkeit sollte auch eine gute Skalierbarkeit des neuen Agentensystems gewährleistet werden, also die Fähigkeit auf gewachsene Anforderungen an das Agentensystem, wie z.B. eine höhere Anzahl von Agenten auf einem Ort, zu reagieren und diese befriedigen zu können. Auch diese Eigenschaft spielt beim kommerziellen Einsatz von Agentensystemen eine wichtige Rolle und wird vom neuen System sichergestellt. Erreicht wird dies durch Ausnutzung der Möglichkeiten, die das Tuxedo-System und die Tandem-Himalaya-Plattform bieten.

Durch einfache Änderungen an der Tuxedo-Konfigurationsdatei können die Zahl der Serverprozesse, welche die Agenten ausführen, verringert oder erhöht werden, die Rechner auf denen diese Prozesse laufen spezifiziert oder neue Orte angegeben werden. Viele Konfigurationsmöglichkeiten können online durchgeführt werden, so können beispielsweise bei einem Rechnerausfall die betroffenen Prozesse auf einen anderen Knoten verschoben werden. Das Tuxedo-System unterstützt den Administrator auch, indem es ihm einige Aufgaben abnimmt. So werden z.B. abgestürzte Prozesse automatisch neu gestartet.

Um gestiegene Softwareanforderungen erfüllen zu können, ist ab einem gewissen Punkt die Erweiterung der Hardware, z.B. um weitere Prozessoren oder Arbeits- bzw. Massenspeicher, vonnöten. Beim Tandem Himalaya K10000 Server sind diese Erweiterungen in sinnvollen Schritten durchführbar. So kann der Server mit bis zu 16 Prozessoren aufgerüstet werden. Reicht auch diese Rechenleistung nicht aus, so können mehrere Systeme vernetzt werden.

6.3 Auslagerung von blockierten Agenten

Da die Zahl der Serverprozesse einer Serverklasse begrenzt ist, sollte die Ausführung von Agenten, die in einen Wartezustand geraten, gestoppt und der Agent in die Message-Queue ausgelagert werden. Das Agentensystem verläßt sich dabei auf den Kooperationswillen des Agenten bzw. dessen Programmierers. Dies vereinfachte die Implementation des Agentensystems erheblich, bürdet dem Agentenprogrammierer aber zusätzliche Arbeit auf. Hält er sich nicht an die Vorgaben, die es dem System ermöglichen, Wartezustände zu erkennen, so hat dies die Blockade des den Agenten ausführenden Serverprozesses zur Folge. Schlimmer sind die Folgen bei mutwilliger Ausnutzung dieser Sicherheitslücke. Durch Denial-of-Service-Angriffe können im Extremfall sämtliche Serverprozesse blockiert, und somit das ganze Agentensystem lahmgelegt werden. Unterstützt der Agentenprogrammierer das System bei der Erkennung

von Wartezuständen, indem er wartende Agententhreads explizit oder implizit beendet, so erfordert dies von ihm eine gewisse Umstellung des Programmierstils.

6.4 Anlehnung an MOLE

Eine weitere Anforderung an das neue Agentensystem betraf die konzeptionelle Anlehnung der agentenspezifischen Funktionen, also Kommunikation und Migration, an MOLE. Diese Anforderung wird trotz der geänderten Architektur weitgehend erfüllt. Dem Agenten stehen im neuen Agentensystem die gleichen Kommunikations- und Migrationsmöglichkeiten wie in MOLE zur Verfügung. Im Bereich der Kommunikation wird der synchrone und asynchrone Nachrichtenaustausch, der entfernte Methodenaufruf und das Konzept der Session unterstützt. Wie auch in Mole, kann der Kommunikationspartner durch seinen Agentennamen oder eine Badge spezifiziert werden. Beim synchronen Nachrichtenaustausch unterscheiden sich die Systeme in der Fehlerbehandlung. Während MOLE hier wie beim asynchronen Nachrichtenaustausch verfährt, meldet das neue Agentensystem aufgetretene Fehler durch Exceptions an den Sender zurück.

Zwar bietet das neue System die gleichen Kommunikationsmöglichkeiten zwischen Agenten, reicht bei deren Effizienz aber nicht an MOLE heran. Dies liegt an einem deutlich spürbaren Zusatzaufwand für die Kommunikation, der nötig ist, da Agenten nicht ständig direkt ansprechbar sind, sondern auch in die Message-Queue ausgelagert werden können.

Die Mobilität von Agenten wird vom neuen System, wie auch in MOLE, durch “schwache” Migration realisiert. Eine Umstellung auf die “starke” Migration würde durch die dafür nötige Möglichkeit, den kompletten Ausführungszustand des Agenten zu ermitteln und wieder einzuspielen, aber, anders als in MOLE, weitere Verbesserungen mit sich bringen. Auf diese wird im nächsten Kapitel näher eingegangen.

6.5 Fazit

Zusammenfassend ist zu sagen, daß im neuen Agentensystem in den wichtigen Bereichen Zuverlässigkeit und Skalierbarkeit deutliche Fortschritte gegenüber MOLE zu verzeichnen sind. Als Preis dafür muß aber eine ineffizientere Kommunikation zwischen Agenten in Kauf genommen werden.

Kapitel 7 Ausblick

In den vorangegangenen Kapiteln wurde das neue Agentensystem vorgestellt. Wie dort an einigen Stellen schon angeklungen ist, gibt es mehrere Ideen die in zukünftigen Arbeiten auf ihre Tauglichkeit geprüft und bei einem positiven Ergebnis realisiert werden könnten. In diesem Kapitel werden diese Ideen näher vorgestellt.

7.1 Ermittlung und Wiedereinspielen des Ausführungszustandes

Die wohl wichtigste Verbesserungsmöglichkeit betrifft die Ermittlung und das Wiedereinspielen des Ausführungszustandes eines Agenten. Dadurch ist nicht nur die Realisierung einer “starken” Migration möglich. Die Auslagerung von wartenden Agenten wäre unproblematischer, da das System nicht mehr auf die Kooperationsbereitschaft des Agenten bzw. dessen Programmierers angewiesen wäre. Auch der Programmierer würde entlastet, da er die für ihn relevanten Teile des Ausführungszustandes vor einer Migration oder Auslagerung des Agenten nicht mehr umständlich in dessen Daten unterbringen und danach wieder restaurieren müßte. Diese Aufgaben würden vollständig vom Agentensystem übernommen, so daß der Programmierer seinen “seriellen” Programmierstil nicht ändern müßte.

Ein weiterer Vorteil ist, daß das Agentensystem die Nutzung der Serverprozesse durch die Agenten mit einem Zeitscheibenverfahren regeln könnte, ähnlich wie moderne Betriebssysteme die Prozessorzeit an die vorhandenen Prozesse verteilen. Dadurch könnten die Ressourcen gerechter verteilt und die Wartezeit eines zur Ausführung bereiten Agenten auf das Freiwerden eines Serverprozesses verringert werden, da Agenten Serverprozesse nicht mehr beliebig lange in Beschlag nehmen könnten, sondern nach einer gewissen Zeit selbst ausgelagert würden. Ein weiterer positiver Effekt wäre, daß Denial-of-Service-Angriffe zwar immer noch stattfinden könnten, ihre Auswirkungen aber deutlich geringer wären. Den normalen Agenten würde zwar durch die Angreifer eine geringere Ausführungszeit zur Verfügung stehen, aber eine vollständige Blockade der Serverprozesse wäre nicht mehr möglich. Um die Lei-

stungsfähigkeit des Systems deutlich zu verringern, müßten sehr viele Angreiferagenten ins Agentensystem eingebracht werden. Dies hätte aber zur Folge, daß der Angriff relativ schnell erkannt würde und evtl. sogar Rückschlüsse auf den Verursacher gezogen werden könnten.

Da die Ermittlung und das Wiedereinspielen des Ausführungszustandes von den meisten Interpretern, so auch von Suns JVM, nicht unterstützt wird, müßten Modifikationen an diesen erfolgen. Voraussetzung dafür ist aber, daß der Ausführungszustand des Agenten vollständig im Datenzustand des Interpreters, und nicht zum Teil in dessen eigenem Ausführungszustand, repräsentiert wird. Aber auch wenn dies möglich ist ergeben sich einige Nachteile. Da die Java-Sprachdefinition noch nicht abgeschlossen ist, müßten bei Erscheinen einer neuen JVM die nötigen Modifikationen erneut durchgeführt werden. Außerdem wäre das Agentensystem auf nicht modifizierten Interpretern nicht mehr lauffähig. Mit dem System müßte der veränderte Interpreter ausgeliefert und installiert werden.

7.2 Leistungsmessungen

Leider blieb keine Zeit mehr um im Rahmen dieser Arbeit umfangreiche Leistungsmessungen durchzuführen. Dies könnte aber in künftigen Arbeiten nachgeholt werden, da sich die aus diesen Messungen ableitbaren Erkenntnisse evtl. zur Steigerung der Effizienz des Agentensystems einsetzen lassen. So könnten Messungen beispielsweise aufzeigen, ob oder in welchen Fällen sich der Zusatzaufwand für die Verwaltung des Caches durch die Sublocation bei der Kommunikation zwischen Agenten lohnt.

7.3 Bereitstellung von Diensten

Gegenwärtig bietet das Agentensystem den Agenten nur die grundlegendsten Dienste, nämlich Kommunikation und Migration, an. Die Realisierung weiterer Dienste könnte folgen. Beispielsweise könnte das Agentensystem seinen Agenten auch Dienste des Tandem-Rechners oder des Tuxedo-Systems zur Verfügung stellen. So könnte in der Tuxedo-Konfigurationsdatei eine zusätzliche Message-Queue eingerichtet werden. Ein Systemagent macht diese Message-Queue dann durch das Anbieten von Zugriffsfunktionen den interessierten Agenten zugänglich.

7.4 Agentennamen

Wie in Kapitel vier schon erwähnt, ist es zur Zeit nicht möglich aus den vom Agentensystem vergebenen Agentennamen den aktuellen Aufenthaltsort des Agenten zu ermitteln. Deshalb ist bei der Spezifikation eines Kommunikations-

partners neben dessen Name auch der aktuelle Aufenthaltsort anzugeben. Die Implementation eines Mechanismus, der den aktuellen Aufenthaltsort aus dem Agentennamen durch Kontaktaufnahme mit dem Heimatort des Agenten ableiten kann, würde die Spezifikation eines Kommunikationspartners vereinfachen.

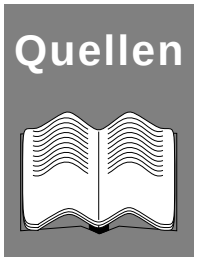
7.5 Verteilte Verarbeitung im LAN

Erst gegen Ende der Entwicklung wurde das Agentensystem auf den K10000-Server portiert. Aus verschiedenen Gründen erfolgte ein Großteil der Entwicklung nicht auf dem Tandem-Rechner, sondern auf PCs bzw. Workstations in einem LAN. Dabei wurde die Message-Queue durch eine Java-Klasse simuliert, ohne dabei jedoch das stabile Speichern oder den transaktionalen Zugriff zu berücksichtigen. Diese Klasse bot queue()- und dequeue()-Operationen an, die über RMI aufgerufen werden konnten. Sublocations konnten von Hand gestartet werden. Zu Beginn ihrer Ausführung entnahmen sie der simulierten Queue durch RMI-Aufruf von dequeue() den nächsten Agenten und führten ihn aus, bis alle Agententhreads beendet wurden. Gegebenenfalls wurde der Agent durch RMI-Aufruf von queue() in die Queue zurückgeschrieben.

Mit dieser Verfahrensweise können beliebig viele Sublocations im gesamten LAN, oder auch darüber hinaus gestartet werden, so daß eine sehr gute Lastverteilung und Skalierbarkeit erreicht wird. Die Zuverlässigkeit des K10000-Servers ist natürlich nicht mehr gegeben.

Diese Vorgehensweise kann als Anregung für weitere Projekte dienen. Beispielsweise ist die Realisierung von Lastmonitoren denkbar, die bei Bedarf zusätzliche Sublocations auf Rechnern des LANs starten.

Literaturverzeichnis



[ACDF] Andrade, Carges, Dwyer, Felts: *The Tuxedo System: Software for Constructing and Managing Distributed Business Applications;* Addison-Wesley, 1996

[BaHoSt] Baumann, Hohl, Straßer: *Mole: A Java Based Mobile Agent System;* Universität Stuttgart, IPVR, 1996

[BHRS] Baumann, Hohl, Rothermel, Straßer: *Mole - Concepts of a Mobile Agent System;* Universität Stuttgart, IPVR, 1997

[BHRRSS] Baumann, Hohl, Radouniklis, Rothermel, Schwehm, Straßer: *ATOMAS: A Transaction-oriented Open Multi Agent-System. Annual Report;* Universität Stuttgart, IPVR, 1994

[GraReu] Gray J., Reuter A.: *Transaction Processing: Concepts and Technologies;* Morgan Kaufman, USA, 1993

[LemPer] Lemay L., Perkins C.L.: *Java 1.1 in 21 Days;* Samsnet, USA, 1997

[Kubach] Kubach,Uwe: *Redesign/Reimplementation der Kommunikationsmechanismen in Mole;* Universität Stuttgart, IPVR, 1997

[Tand95a] Tandem Manual PartNumber 115734: *Introduction to Tandem NonStop Systems;* Tandem Computer Inc., 1995

[Tand95b] Tandem Manual PartNumber 111157: *Introduction to the NonStop Himalaya K10000/K20000;* Tandem Computer Inc., 1995

[Tand95c] Tandem Manual PartNumber 110444: *NonStop Tuxedo System Programmer's Guide;* Tandem Computer Inc., 1995

[Tand95d] Tandem Manual PartNumber 110440: *NonStop Tuxedo System Application Development Guide;* Tandem Computer Inc., 1995

[Tand95e] Tandem Manual PartNumber 125335: *Introduction to Tandem NonStop Transaction Processing;* Tandem Computer Inc., 1995

[Tand95f] Tandem Manual PartNumber 103929: *NonStop Tuxedo System Administration Guide;* Tandem Computer Inc., 1995

[Tand96a] Tandem Manual PartNumber 128187: *Open System Services User's Guide;* Tandem Computer Inc., 1996

[Weber] Weber M. *Teamwork mit Methode: Ein Chat-Server mit Javas RMI-Interface;* c't magazin für computertechnik, S.348ff, Heise, Heft 11/1997

Hiermit versichere ich, daß ich diese Arbeit selbstständig verfaßt und bei der Erstellung nur die angegebenen Hilfsmittel verwendet habe.

Michael Bader