Prüfer:	Prof. Dr. rer. nat. Kurt Rothermel
Betreuer:	Dipl. Inform. Joachim Baumann
Begonnen am:	15. November 1996
Beendet am:	14. Mai 1997
CR-Klassifikation:	C.2.2, C.2.4, D.4.4, D.4.5

Diplomarbeit Nr. 1472

Terminierung und Waisenerkennung in einem System mobiler Software-Agenten

Bernhard Beck

Danke

Wenn in dieser Arbeit von "uns" oder "wir" die Rede ist, so sind damit normalerweise mein Betreuer Joachim Baumann und ich, sowie die Eventgruppe der Abteilung Verteilte Systeme gemeint.

In vielen Diskussionen wurden Ideen und Konzepte durchgesprochen und verfeinert. Es hat Spaß gemacht mit Euch zu arbeiten.

Auf diesem Weg auch Dank an alle, die mich bei dieser Arbeit mit Nahrung aller Art unterstützt haben (Tips und Hinweise, Kaffee und Tee, Gummibärchen und Kekse, ...).

Anmerkung zur Onlineversion

Die vorliegende Onlineversion beinhaltet denselben Text wie die gedruckte Version der Diplomarbeit. Es wurden lediglich die Schriftarten auf Standard-Postscript angepaßt.

Wegen Beschränkungen meines Druckertreibers kann die Ausarbeitung leider nur komplett gedruckt werden. Der Ausdruck sollte möglichst auf einem Postscriptdrucker mit viel Speicher erfolgen. Zur Not kann auch Ghostscript verwendet werden.

Inhaltsverzeichnis

Erster Teil: Einführung 1

Was erwartet uns in dieser Arbeit? 2

Begriffe und Definitionen 3
- Events und Eventmanagement 3
  - Das Nachrichtenparadigma - ein kleiner Ausflug
  - Der Event
  - Der Eventchannel
  - Der Eventmanager
  - Modell des Eventmanagers
- Das Problem der (un)einheitlichen Zeit 6
- Broadcast und Multicast 7
- Multicastsemantiken 8
  - Reliable Multicast
  - Atomic Multicast
  - Causal Multicast
  - Totalordnung von Multicasts
- Convergecast 9

Agenten und Agentensysteme 11
- Der Agent 11
- Der mobile Agent 11
- Agentensysteme 12
- MOLE 13
- Agentengruppen 14
  - Der Group Coordinator (GC)
  - Die Gruppenbedingung
  - Gruppensemantiken

Agentenszenarien 16
- Reiseplanung 16
- WWW-Recherche 16
- Lokale Beobachter 17
- Literaturrecherche im Internet 17
- Elektronische Märkte 18
- Folgerungen 18

Zweiter Teil: Anforderungen an das Eventmanagement 19

Grundlegende Anforderungen 20
- Verteilung 20
- Die Schnittstelle zum Eventmanager 20
- Eventchannels 21
- Authentifikation und Autorisierung 22
- Weitere Anforderungen 23

Charakteristik von Eventchannels 24
Kriterien 24
Gruppengröße
Gruppenanzahl
Gruppenausdehnung
Mitgliederfluktuation
Gruppenaktivität

Terminierung 29
Explizite Terminierung von Agenten 29
Ein Beispiel
Ein Protokoll zur Lösung des Problems
Was gibt es denn da für besondere Anforderungen?
Verfeinerung
Kosten
Implizite Terminierung mit Agentengruppen 31
Anforderungen an den Eventmanager
Folgerungen 32

Waisenerkennung 33
Waisen 33
Wer bestimmt Waisen?
Die oberste Ebene
Wann können Waisen entstehen? Ein Beispiel
Aktive Waisenerkennung 35
Ein mögliches Protokoll
Sicherheit
Bestimmung von Timeout und TimeoutCheck
Einfluß von Migration
Anforderungen an den Eventmanager
Und was kostet das?
Passive Waisenerkennung 39
Ein Protokoll
Sicherheit
Verbesserungen
Anforderungen an den Eventmanager
Kosten
Skalierbarkeit
Waisenverhinderung: Das Energiekonzept 43
Prinzip
Ein Protokoll
Was bringt es uns?
Beurteilung
Folgerungen 45

Eigenschaften von Events 46
Eventarten 46
Speicherung von Events 46
Der Speicherort von Events
Lebensdauer von Events 48
Reihenfolge von Events 49
Terminierung und Waisenerkennung
Synchronisierung

Mobilität von Objekten 51
Mobile Consumer 52
Ein Lösungsansatz ...
... und die Folgen
Mobile Supplier 53
Binden von Objektreferenzen 54
Berechnung einer Identifikation
Ein Keks vom Krümelmonster
Agentennamen
Fazit

Einfügen neuer Knoten 57
Wo liegt das Problem? 57
Lösungsansätze 57
Statische Methoden
Internet-Multicast
Bewertung
... und wie findet ein lokaler Eventmanager seinen Dämon? 59

Fehlerbehandlung 60
Ausfall einer Übertragungsstrecke ("Link") 60
Ausweichen möglich
Netzpartitionierung
Ausfall eines Knotens 61
Ausfall des ganzen Rechners
Ausfall des Eventdämons
Re-Konfiguration des spannenden Baums 62
Umgehen
Neukonfiguration
Eine andere Methode zur Re-Konfiguration

Anforderungen auf einen Blick 68

Dritter Teil: Eventmanager 69

Der CORBA Event Service 70

Basic Building Blocks 72
IP Multicast und statisches Multicastrouting 72
Multicasting im Internet
Das Internet Group Multicast Protocol (IGMP)
Jenseits des Multicastrouters
Bewertung
Dynamisches Multicastrouting 76
Restricted Broadcast
Nearest Insertion
Mobile Object Multicast Protocol (MOMP)
Vorteile und Nachteile dynamischen Multicastroutings
Kausale Ordnung von Events 83
Der Algorithmus
Überlegungen zur Optimierung
Löschen von Events aus BUF
Aufbau eines minimal spannenden Baums 86
Der Algorithmus
Kriterium zur Neuberechnung des spannenden Baums
Message Queues 89
Funktionsprinzip und Eigenschaften
Implementierungen

Gruppenkommunikation in Amoeba 91
Kommerzielle Produkte 92
OrbixTalk 92
TIBCO Rendezvous 93
Bewertung 94

Vierter Teil: Implementierung 95

Entwurf der Kommunikationsstrukturen 96
Der Link 97
Die Linkfactory 98
Der Linkserver 98
Der MultiLink 98
Events und Event-IDs 99
Die Protokollschnittstelle 100
Der PingClient 100
Der Channel-Client 100
Der StaticMSB-Client 100
Der RB-Client 101
Der MOMP-Client 101
Anbindung an MOLE 102
EventDämon und lokaler Eventmanager 102
Lokaler Eventmanager und MOLE 103
Ausblick 104

Fünfter Teil: Anhänge 107

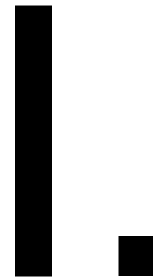
Logische Uhren 108
Algorithmen für dynamischen Multicast 110
Restricted Broadcast 110
Mobile Object Multicast Protocol 112
Nearest Insertion 114
Literatur 117
Glossar 121

Verzeichnis der Abbildungen

I.1	Modell des Eventmanagers	5
I.2	Vergleich von verbindungsorientierter und gruppenorientierter Kommunikation	8
I.3	Darstellung eines Convergecasts	10
II.1	Erläuterung zu Formel (II.8), Mitgliederfluktuation	28
II.2	Wann ist der Makleragent eine Waise?	34
II.3	Pseudocode Waisenerkennung	36
II.4	Die Problematik mobiler Consumer	53
II.5	Lokale Umkonfiguration des Baums bei Ausfall von Knoten	63
III.1	Der CORBA-Eventchannel	70
III.2	Push- und Pull-Modell bei CORBA	70
III.3	Mögliche Fehlerfälle	81
IV.1	Kommunikationsstruktur des Eventmanagers	96

Verzeichnis der Formeln

(II.1)	relative Größe einer Knotengruppe	25
(II.2)	Gesamtanzahl Agentengruppen	25
(II.3)	geographische Gruppenausdehnung allgemein	26
(II.4)	geographische Gruppenausdehnung spannender Baum	26
(II.5)	Berechnung des Kantengewichts für spannenden Baum	26
(II.6)	Mitgliederfluktuation, Beitreten	27
(II.7)	Mitgliederfluktuation, Verlassen	27
(II.8)	Mitgliederfluktuation, Umziehen	27
(II.9)	Aktivität der Gruppe	28
(II.10)	Aktivität eines Suppliers	28
(II.11)	Kosten eines Events in einem spannenden Baum	38
(II.12)	Kosten aktiver Waisenerkennung	39
(II.13)	Kosten passiver Waisenerkennung, Basisalgorithmus	41
(II.14)	Kosten einer Nachricht für passive Waisenerkennung, Direkte Variante	42
(II.15)	Kosten passiver Waisenerkennung, Direkte Variante	42
(II.16)	Kosten passiver Waisenerkennung, Channelvariante	42
(II.17)	Bestimmung der Qualität einer Umleitung bei Knotenausfall	64
(II.18)	Kosten für Umschalten auf neuen Baum bei Neukonfiguration	66
(II.19)	Kosten für Umschalten auf neuen Baum bei Neukonfiguration	66



I. Einführung

Im ersten Teil der Diplomarbeit werden grundlegende Begriffe und Definitionen aus dem Bereich der verteilten Systeme kurz besprochen.

Weiterhin werden Agentensysteme im allgemeinen und das Agentensystem MOLE, das dieser Arbeit zugrundeliegt, im besonderen vorgestellt. Einige Anwendungsszenarien für mobile Agenten runden den ersten Teil ab.

I.1. Was erwartet uns in dieser Arbeit?

Das Problem der Terminierung von Agenten und der Waisenerkennung, dem Finden von Agenten, die keine sinnvolle Aufgabe mehr erledigen, ist eine der großen Herausforderungen in Systemen mobiler Software-Agenten. Klassische Terminierungsstrategien, wie sie für den Remote Procedure Call entwickelt wurden oder Verfahren, um festzustellen, ob ein verteilt ausgeführter Algorithmus terminiert ist, können nicht ohne weiteres auf den Kontext eines Agentensystems übertragen werden. Die Beziehungen zwischen den Agenten können um einiges komplexer als bei klassischen Client-Server-Systemen sein. Insbesondere ist es möglich, daß Agenten noch sinnvolle Aufgaben ausführen, obwohl ihre Eltern bereits seit langem terminiert sind. Daher müssen neue Strategien eingeführt werden, um solche elternlosen Agenten von echten Waisen, die keine sinnvolle Aufgabe mehr haben, unterscheiden zu können.

In dieser Arbeit entwickeln wir die Infrastruktur eines Systems für anonyme Kommunikation in Form von Ereignissen ("Events"). Im ersten Teil stellen wir Konzepte von Eventkommunikation und Agentensystemen vor. Wir gehen dabei besonders auf Probleme ein, die in weit verteilten Systemen auftreten. Zum Abschluß des ersten Teils zeigen wir mehrere Anwendungsszenarien für Agentensysteme auf.

Im zweiten Teil der Arbeit untersuchen wir, welche grundsätzlichen Anforderungen die Verwendung von Events in einem weltweit verteilten System an den Eventmanager stellt. Für die Kommunikation innerhalb des Eventmanagers verwenden wir spannende Bäume über den beteiligten Rechnerknoten. Darauf aufbauend werden die zusätzlichen Anforderungen von Protokollen und Algorithmen zur Terminierung und Waisenerkennung herausgearbeitet. In den restlichen Kapiteln des zweiten Teils beschäftigen wir uns mit der Behandlung von Fehlern, den Eigenschaften von Events und stellen die speziellen Aufgabengebiete vor, die durch mobile Eventnutzer auftreten.

Im dritten Teil beschreiben wir Lösungskonzepte und Bausteine für das Eventmanagement. Wir betrachten dazu kurz mehrere kommerzielle Eventmanager, sowie allgemeine Konzepte wie zum Beispiel Multicastkommunikation. Die Implementierung und Integration des Eventmanagers in das Agentensystem beschreiben wir im vierten Teil.

Im Anhang sind die meisten der vorgestellten Algorithmen formal dargestellt. Weiterhin findet sich dort das Literaturverzeichnis und ein Glossar zu wichtigen Begriffen und Abkürzungen aus dem Themengebiet dieser Arbeit. Abkürzungen werden bei ihrer ersten Verwendung im Text, sowie im Glossar erklärt.

I.2. Begriffe und Definitionen

Das Kapitel beschreibt einige allgemeine Problemkomplexe, sowie Definitionen und Begriffe aus dem Bereich der verteilten Systeme, die in dieser Arbeit von grundlegender Bedeutung sind und einer ausführlicheren Erläuterung bedürfen.

I.2.1. Events und Eventmanagement

In diesem Abschnitt werden die Konzepte von Events und Eventmanagern vorgestellt [Orbix96]. Damit erstellen wir das Modell des Eventmanagementsystems, das dieser Arbeit zugrundegelegt wird.

Das Nachrichtenparadigma - ein kleiner Ausflug

Es vergeht keine Sekunde, zu der nicht Informationen ausgetauscht werden. Zwischen Menschen, zwischen Computern, weltweit und lokal. Abgesehen von einem Gespräch in einer kleinen Gruppe, gibt es bei jeder Kommunikation einen Punkt, an dem das gewählte Kommunikationsmittel nicht mehr sinnvoll ist. Wir verlassen uns normalerweise auf die gesprochene Sprache. Aber: egal wie laut man schreit, die Menschenmenge muß nur groß oder verteilt genug sein, damit nicht mehr alle Hörer die Sätze verstehen können. Der Mensch hat viele Möglichkeiten geschaffen, Informationen verbreiten: Telefon, Briefe, Lautsprecher, EMail, Fax, Fernsehen, ... All diese Möglichkeiten basieren auf der Entkopplung von Sprecher und Publikum. Der Sprecher weiß nicht unbedingt, wer ihm zuhört und braucht es, insbesondere bei den Massenmedien, auch nicht zu wissen.

Kommunikation zwischen Computern folgt, wie menschliche Kommunikation auch, bestimmten Regeln und Konventionen. Ein Prozeß sendet Daten und ein anderer Prozeß bestätigt jedes einzelne empfangene Byte. Ein Übertragungsprotokoll ist für die Erkennung fehlender oder zerstörter Daten zuständig. Die Kommunikation zwischen den zwei Teilnehmern ist also zuverlässig. Dieser Ansatz funktioniert aber nicht mehr, wenn die Anzahl der Empfänger einer Nachricht unbekannt ist oder es zu viele sind, als daß dem Sender zugemutet werden könnte, mit jedem Empfänger direkt zu kommunizieren. Daher ist es besser, Sender und Empfänger einer Nachricht nicht direkt miteinander kommunizieren zu lassen, sondern ein Zwischenmedium einzuführen. Was für das tägliche menschliche Leben gut ist, kann in der Informatik nur billig sein.

Unter Verwendung der Methode des indirekten Nachrichtenaustauschs können beliebige Mengen von Sendern und Empfängern miteinander kommunizieren. Wir benötigen nur noch ein System, das die Zustellung der Nachrichten zuverlässig erledigt.

Wir benötigen einen Eventmanager.

Der Event

Event heißt wörtlich übersetzt Ereignis. Analog zu objektorientierten Sprachen und Systemen ist ein Event in unserem Modell die Nachricht, daß ein bestimmtes Ereignis eingetreten ist oder eintreten soll. Eine ereignisorientierte Sicht der Dinge erleichtert den Entwurf einer Applikation ungemein, da sich Sachverhalte der realen Welt unter Verwendung von Events oft einfacher beschreiben lassen als mit prozeduralen Beschreibungen. Der Event ist ein (persistentes) Objekt, das beliebige Inhalte transportiert. Der Eventtyp beschreibt diese Inhalte, entspricht also der Klasse des Eventobjekts.

Der Eventchannel

Events werden von Eventerzeugern ("Supplier") erzeugt und von Eventverbrauchern ("Consumer") empfangen. Prinzipiell darf jedes Objekt, das den Eventmanager verwenden will, als Consumer oder Supplier auftreten. Supplier und Consumer kommunizieren über einen gemeinsamen Kommunikationskanal, den wir "Eventchannel" nennen. Der Eventchannel garantiert Dienstmerkmale, die im zweiten Teil der Diplomarbeit erarbeitet werden. Supplier und Consumer müssen sich auf einen gemeinsamen Eventchannel geeinigt und dort angemeldet haben, bevor sie mit der Kommunikation beginnen. Pro Eventchannel dürfen beliebig viele Supplier und Consumer auftreten. Es handelt sich dabei um eine n:m-Kommunikation.

Der Eventmanager

Der Eventchannel ist die logische Sicht zur Verteilung von Events zwischen Suppliern und Consumern und beschreibt die Dienstmerkmale. Der Eventmanager stellt diese Funktionalität zur Verfügung und ist für die korrekte Abarbeitung, Fehlerkorrektur, Verwaltung und die Einhaltung aller gestellten Bedingungen zuständig.

Modell des Eventmanagers

Abbildung I.1 zeigt unser Modell eines Eventmanagers. Jede Anwendung, die die Dienste des Eventmanagers verwenden möchte, implementiert einen lokalen Eventmanager. Dieser bildet den Zugangspunkt zum Eventmanagementsystem und kommuniziert mit einem Eventdämon, der für die optimale, daß heißt möglichst ressourcenschonende, Verteilung der Events zuständig ist. In einer optimalen Konfiguration ist pro beteiligtem Rechner ein Dämonprozeß vorhanden, der mit anderen Dämonprozessen kommuniziert. Die Anwendungen kommunizieren über ihren lokalen Eventmanager unter Verwendung lokaler Kommunikationsmechanismen mit dem Eventdämon. Man kann sich allerdings leicht vorstellen, bereits für diesen Schritt eine verbindungsorientierte Kommunikation zwischen lokalem Eventmanager und Eventdämon einzuführen. Dies hätte den Vorteil, daß auch Anwendungen auf Rechnern, auf denen kein Eventdämon aktiv ist (zum Beispiel "Personal Digital Assistants",PDAs), an der Eventkommunikation teilnehmen können. Ihr lokaler Eventmanager kommuniziert dann nicht mit dem lokalen, sondern mit einem entfernten Eventdämon.

Wie die Events zwischen den Eventdämonen verteilt werden, wird in den folgenden Kapiteln dieser Diplomarbeit erarbeitet. Das Konzept der lokalen Eventmanager verwenden wir hauptsächlich dazu, um den Eventmanager von den Anwendungen zu trennen und so eine möglichst universelle Nutzung zu ermöglichen. Natürlich liegt das Hauptaugenmerk zunächst auf der Nutzung durch MOLE, dem Agentensystem des IPVR. Andererseits ist es durch die eindeutige Schnittstelle einfacher, einen kommerziellen Eventmanager (zum Beispiel die Orbix Event Services) für die Verteilung der Events zu verwenden.

Ferner nehmen wir an, daß die lokale Interprozeßkommunikation der Anwendungsprozesse mit dem Eventdämon im Verhältnis zur Netzwerkkommunikation sehr billig ist. Wenn im folgenden von Eventmanager die Rede ist, so ist damit die Gesamtheit aller Prozesse und Kommunikationspfade gemeint, die für das Management und Verteilen von Events zuständig sind.

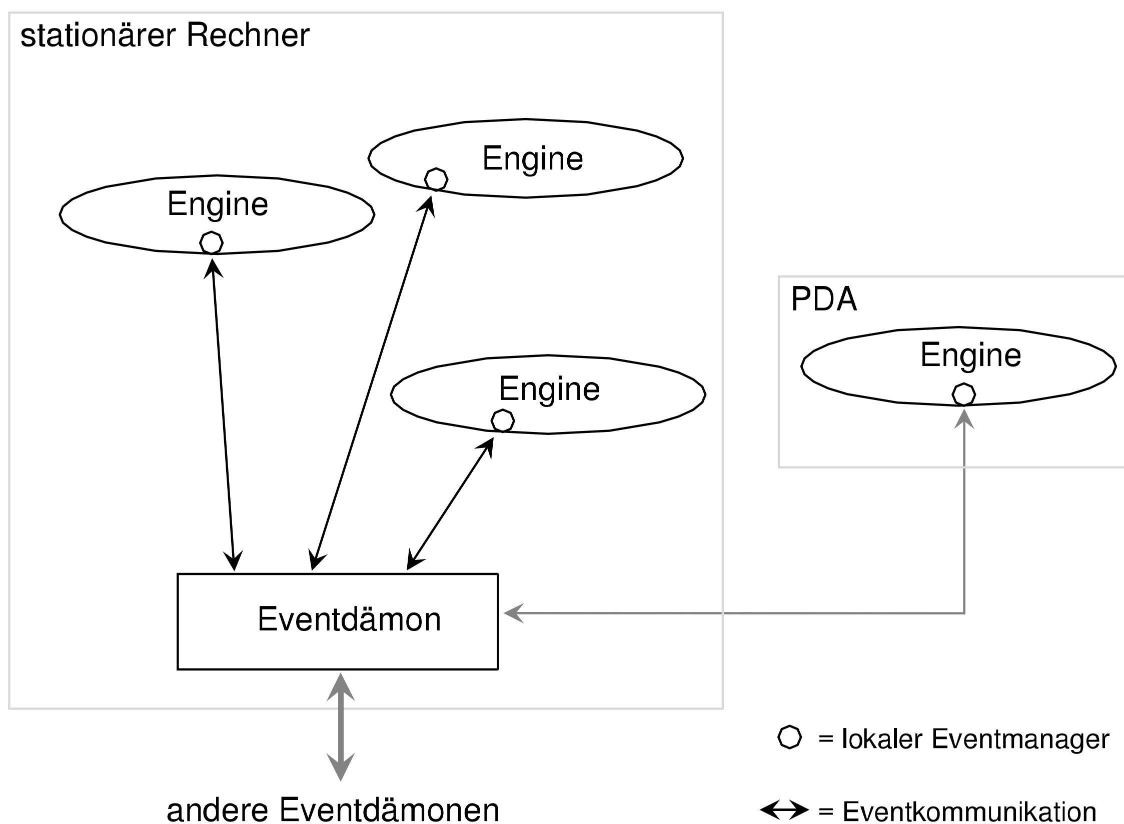


Abb. I.1: Modell des Eventmanagers und Anbindung an das Agentensystem MOLE. Optimierte Kommunikation findet nur zwischen Eventdämonen statt.

Ohne allzuweit auf die folgenden Kapitel vorzugreifen, noch einige Anmerkungen zu Implementierungsaspekten des Eventmanagers. Die Kommunikation zwischen den beteiligten Rechnern soll möglichst effizient und ressourcenschonend erfolgen. Die Bandbreite von Weitverkehrsnetzen ist auch heute noch eine teure Ressource. Zur Kommunikation zwischen den einzelnen Rechnern, beziehungsweise deren Subnetzen, werden daher oft spannende Bäume verwendet, auf deren Kanten die Events an alle beteiligten Knoten verteilt werden. Wenn bei der Konstruktion der spannenden Bäume zusätzlich die

Kommunikationskosten zwischen zwei Knoten in Betracht gezogen werden, können die Bäume auf minimale Kommunikationskosten optimiert werden. So wird das Versenden eines Events zwischen allen interessierten Knoten so billig wie nur irgendwie möglich. Es kann gezeigt werden, daß spannende Bäume in Hinblick auf Durchsatz und Skalierbarkeit für Multicast-Kommunikation am besten geeignet sind [Levine97]. Wir beschränken uns beim Design des Eventmanagers daher auf Protokolle und Verfahren, die spannende Bäume verwenden.

Bei *n* Knoten sind zum Verteilen eines Events in einem spannenden Baum minimal *n*-1 Übertragungen nötig. Dem steht aber der potentiell sehr hohe Aufwand für die Pflege des Baums gegenüber.

> *Es existieren Algorithmen, die einen spannenden Baum über einem planaren Graphen in* $O(n \log_2 n)$ *konstruieren, indem sie die Knoten und ihre anliegenden Kanten nach Gewicht sortieren. Allerdings können diese nicht verteilt realisiert werden, sondern benötigen einen zentralen Knoten, der den spannenden Baum erstellt und dann an alle anderen Knoten verteilt.*

Wir werden uns in dieser Diplomarbeit mit dieser Problematik auseinandersetzen und Kriterien festlegen, anhand derer entschieden werden kann, wie spannende Bäume am effizientesten für die Verteilung von Events eingesetzt werden können.

I.2.2. Das Problem der (un)einheitlichen Zeit

Bei vielen Aufgabenstellungen ist es wichtig festzustellen, ob ein Event vor einem anderen eingetreten ist. Ein verteiltes System besteht aus autonomen Rechnern mit jeweils einer eigenen Uhr, die über Nachrichtenaustausch kommunizieren.

Die Angabe, daß ein Event *a* vor einem anderen Event *b* eingetreten ist, bedeutet, daß der Zeitpunkt des Eintretens von *a* vor dem Zeitpunkt des Eintretens von *b* liegt. Dabei wird implizit angenommen, daß die Zeitpunkte an derselben Uhr abgelesen werden. Wenn die Zeitpunkte an unterschiedlichen Uhren abgelesen werden, können wir nur durch das Ablesen der Uhren nicht sicher feststellen, ob ein Event vor einem anderen aufgetreten ist, da die Uhren unterschiedliche Zeiten anzeigen können. Da wir in einem verteilten System keine globale Uhr haben, können wir die "happened-before"-Relation nicht so einfach auf Basis der physischen Rechneruhr definieren.

Auf manchen Systemen läßt sich die Rechneruhr auch nicht auf die Millisekunde genau ablesen (zum Beispiel bei PCs nur auf etwa 50 ms genau). Das bedeutet, daß Events, die sehr kurz nacheinander auftreten, denselben Zeitstempel erhalten und somit nicht unterscheidbar wären. Hinzu kommen Verzögerungen des Netzwerks, die sich zudem auch noch laufend durch wechselnde Last ändern können. Mit einem einfachen "Ausmessen" und Aufaddieren der Netzwerkverzögerung zur Bestimmung des Zeitunterschieds zweier Uhren ist es also auch nicht getan.

> *Nichtsdestotrotz existieren Protokolle zur Sychronisation von Rechneruhren (zum Beispiel das "Network Time Protocol", [Mills92]), die aber zum Teil sehr aufwendig und teuer sind.*

Die Verwendung der Rechneruhr ist aber auch gar nicht nötig, da uns eigentlich nur interessiert, in welcher Reihenfolge die Events aufgetreten sind. Die Uhrzeit ist unerheblich, da die Laufzeit der Nachrichten durch Verzögerungen des Netzwerks unterschiedlich beeinflußt werden. Lamport zeigt, daß ein System von Zählern ("logische Uhren") genug Informationen für die Bestimmung der Reihenfolge von Nachrichten liefert (siehe Anhang A).

Wir werden uns im zweiten Teil nochmal mit der Reihenfolge von Events beschäftigen und untersuchen, welche Reihenfolgebeziehungen für die Aufgabenstellung dieser Arbeit überhaupt notwendig sind.

I.2.3. Broadcast und Multicast

Im Netzwerkbereich versteht man unter einem Broadcast das Versenden einer Nachricht an alle Knoten des Netzwerks, während ein Multicast eine Nachricht an eine bestimmte Gruppe von Knoten sendet. Ein Sender sendet an an viele Empfänger, deren Adressen dem Sender nicht unbedingt bekannt sein müssen. Beide Mechanismen verwenden dazu implizit oder explizit Informationen über die Topologie des Netzes, um die Kommunikationskosten zu minimieren. Daher sind Broadcast und Multicast üblicherweise in der Netzwerkschicht implementiert, die die Routinginformationen in den Routern speichert und auf dem aktuellen Stand hält. Ein Broadcast entspricht einem Multicast an alle Rechner eines Netzes und ist vom Konzept her ein Spezialfall von Multicast. Wir verwenden daher im folgenden nur noch den Begriff Multicast. Informationen über Broadcasts im Internet finden sich unter anderem in [Mogul84].

> *Broadcast läßt sich prinzipiell sehr viel einfacher implementieren als Multicast. Allerdings ist ein effizienter und ressourcenschonender Broadcast in der Implementierung beinahe so komplex wie ein Multicast.*

Multicasts sind immer dann nützlich, wenn Ressourcen verwendet werden sollen, die entweder zu kurzlebig, zu abstrakt oder auch einfach nur zu zahlreich sind, als daß man ohne weiteres den Überblick bewahren könnte. Beispielsweise setzen verteilte, replizierte Datenbanken oft einen Multicastservice des Netzes voraus, um die für die Aktualisierung von Replikaten benötigte Netzwerkbandbreite zu verringern. Multicast vereinfacht auch die Aktualisierung der Replikate, wenn man nicht weiß, wie viele Replikate existieren. Die Funktionalität von Events in verteilten Applikationen läßt mit einem vom Netzwerk angebotenen Multicastmechanismus einfacher und ressourcenschonender implementieren, als mit verbindungsorientierter Kommunikation zwischen jedem beteiligtem Knoten.

Zu beachten ist, daß die meisten in dieser Arbeit beschriebenen Multicastalgorithmen auf Netzwerkebene zwischen Rechnern arbeiten. Über die Adressierung von Anwendungsmodulen, die auf den einzelnen Rechnern laufen, ist damit noch nichts ausgesagt. Dies muß in einer höheren Schicht erledigt werden und betrifft beispielsweise die bereits erwähnten Eventchannels. Mit Multicast können Eventchannels implementiert werden.

Abbildung I.2 stellt den Ansatz der verbindungsorientierten Kommunikation der Multicast-Kommunikation im Schichtenmodell gegenüber. Die Darstellung orientiert sich an der Protokollhierarchie des Internets. Auf beiden Seiten sind die normalerweise im Internet verwendeten Protokolle angegeben. Während die Transportschicht der verbindungsorientierten Kommunikation inzwischen gut erforscht und standardisiert ist, sind äquivalent effiziente Ansätze im Bereich der Multicasts noch Gegenstand der aktuellen Forschung. Für die Unterstützung der Multicastfunktionalität ist eine weitere Schicht nötig, die die zur Verwaltung der Multicaststruktur nötigen Protokolle beherbergt. Das Internet Group Multicast Protocol (IGMP) beschreibt die Schnittstelle zu einer vom Netz angebotenen Multicastfunktionalität. Die eigentlichen Multicastpakete werden dann über "normale" UDP-Nachrichten, die von Multicast-Routern besonders behandelt werden, versendet. Das Mobile Object Multicast Protocol (MOMP) wird in dieser Diplomarbeit entwickelt und bietet eine zu IGMP ähnliche Funktionalität für mobile Objekte an.

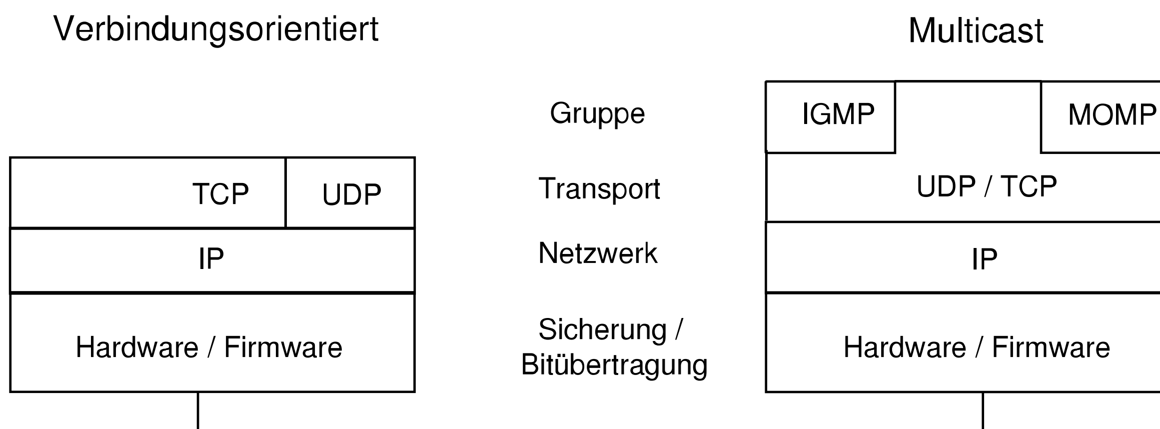


Abb. I.2: Vergleich von verbindungsorientierter und gruppenorientierter Kommunikation zwischen Prozessen.

I.2.4. Multicastsemantiken

In diesem Abschnitt beschreiben wir Semantiken der Auslieferung von Nachrichten eines Multicasts bei den Empfängern [Jalote94]. Obwohl Events auf einer anderen logischen Ebene im Schichtenmodell angesiedelt sind, können dieselben Semantiken angewendet werden. Im zweiten Teil der Arbeit werden sie zur Beschreibung der Anforderungen an einen Eventmanager verwendet.

Reliable Multicast

Der Reliable Multicast (= zuverlässiger Multicast) garantiert, daß eine Nachricht bei jedem Zielknoten ausgeliefert wird.

Atomic Multicast

Beim Atomic Multicast wird eine Reihenfolge der einzelnen Nachrichten festgelegt und garantiert, daß alle Nachrichten an jedem Knoten in derselben Reihenfolge ausgeliefert werden. In welcher Reihenfolge dies geschieht, kann vom Algorithmus frei entschieden werden.

Causal Multicast

Der Causal Multicast arbeitet ähnlich dem Atomic Multicast, mit einer Erweiterung: Die Reihenfolge der Nachrichten bei der Auslieferung wird nun über kausale Zusammenhänge der Nachrichten bestimmt. Durch die kausalen Abhängigkeiten ergibt sich eine Partialordnung der gesendeten Nachrichten.

Es gibt Anwendungen, zum Beispiel in verteilten Datenbanksystemen, bei denen zwischen zwei Multicastnachrichten ein kausaler Zusammenhang besteht. Angenommen die Änderungsoperation einer verteilten Datenbank ist mit Multicasts implementiert. Ein Client ändert einen Datensatz ($Wert = Wert \times 2$) und informiert einen anderen Client darüber, der nun ebenfalls diesen Datensatz ändert ($Wert = Wert + 1$). Die beiden Änderungen stehen in eine kausalen Zusammenhang und müssen in dieser Reihenfolge ausgeführt werden, damit das korrekte Ergebnis $Wert = Wert \times 2 + 1$ errechnet wird. Es wäre fatal, wenn zunächst die Änderung des zweiten Clients wirksam wird und dann die Änderung des ersten. Lamport definiert zum Beschreiben von möglichen kausalen Abhängigkeiten die "Happened-before"-Relation (siehe Anhang A).

Totalordnung von Multicasts

Es ist bei jeder Semantik möglich, Nachrichten auch global über mehrere Prozesse hinweg zu ordnen, wenn die sonstigen Rahmenbedingungen eingehalten werden. Reliable und Atomic Multicast stellen in dieser Hinsicht keine besonderen Anforderungen, da sie entweder keine Ordnung haben oder eine beliebige. Eine Totalordnung von Causal Multicasts muß die kausalen Abhängigkeiten von Nachrichten berücksichtigen. Es wird dann nur noch für Nachrichten, die in keiner kausalen Abhängigkeit stehen, entschieden, welche Ordnung sie untereinander und im Verhältnis zu allen anderen Nachrichten haben.

I.2.5. Convergecast

Der Convergecast ist ein Multicast auf einem spannenden Baum in "Gegenrichtung", daß heißt ausgehend von den Blättern in das Zentrum des Baums [Wall80]. Ein Convergecast wird normalerweise als Reaktion auf einen Multicast ausgeführt. Jedes Blatt des Baums sendet dazu die Nachricht auf seiner einzigen Kante in das Innere des Baums. Alle innenliegenden Knoten sammeln bei n Kanten n-1 Nachrichten und senden dann selber eine Nachricht über die verbliebene Kante weiter. Es existiert im allgemeinen genau eine Kante zwischen zwei Knoten, an der sich die nach innen gesendeten Nachrichten kreuzen. Diese Kante ist das Zentrum des spannenden Baums. Die beiden anliegenden Knoten müssen nun anhand einer eindeutigen Regel festlegen, welcher der zwei der Zentralknoten ist. Die

Regel muß auf beiden Knoten gleich beantwortet werden und darf nicht von dynamisch veränderlichen Eigenschaften des Baums oder des Netzes abhängen. Eine naheliegende Regel ist, daß der Knoten mit der kleineren Netzwerkadresse der Zentralknoten ist. Netzwerkadressen von Rechnerknoten sind immer eindeutig und lassen sich leicht in eine aufsteigende Totalordnung bringen, so daß die Bestimmung des "kleineren" Knotens einfach ist. In Abbildung I.3 sind die Nachrichten eines Convergecast dargestellt.

Zu beachten ist, daß der Convergecast nicht unbedingt an dem Knoten konvergieren muß, von dem der auslösende Multicast ausging! Die Position der zentralen Kante bestimmt sich lediglich aus der realen Laufzeit der Nachrichten von den Blätterm in das Innere des Baums. Eine angenehme Eigenschaft des Zentralknotens ist, daß die Verzögerung zu allen Blättern des Baums in etwa gleich ist (zumindest zum Zeitpunkt des Convergecasts). Das heißt, es kann eine obere Grenze G für die Laufzeit einer Nachricht vom Zentralknoten zu jedem Blatt angegeben werden. Damit kann auch eine obere Grenze für die Laufzeit einer Nachricht über den spannenden Baum von jedem Knoten zu jedem beliebigen anderen Knoten angegeben werden, indem die Nachricht über den Zentralknoten gesendet wird. Die obere Grenze liegt dann bei $2 \times G + \varepsilon$ (ε für die Verarbeitungszeit im Zentralknoten).

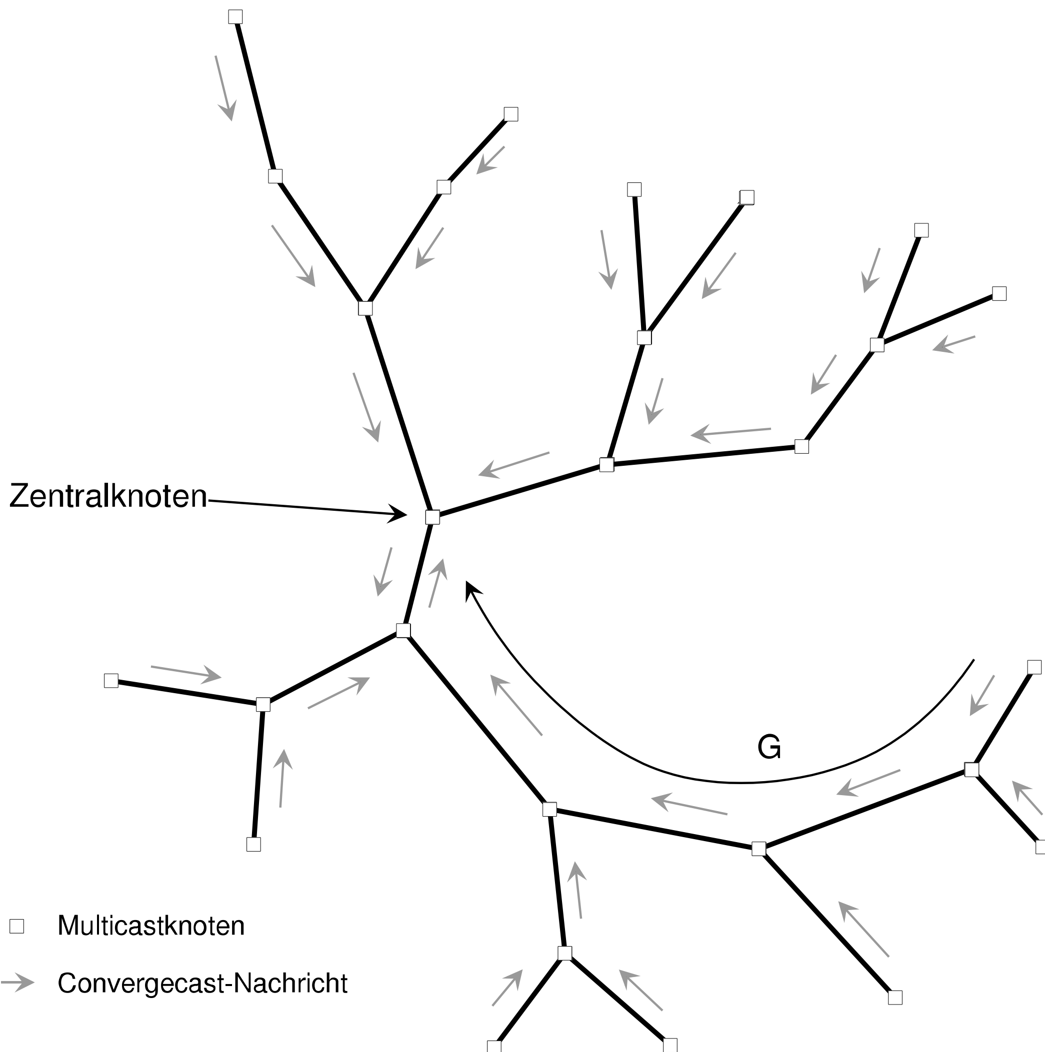


Abb. I.3: Darstellung eines Convergecasts über einem spannenden Baum. Die Nachrichten treffen sich am Zentralknoten in der "Mitte" des Baums.

I.3. Agenten und Agentensysteme

In diesem Kapitel stellen wir Konzepte von Agenten und Agentensystemen vor.

I.3.1. Der Agent

In der Literatur und aktuellen Veröffentlichungen herrscht einige Konfusion über den Begriff "Agent". Es hat den Anschein, daß jeder Autor eine eigene Definition verwendet. So gibt es im Netzwerkmanagementbereich Agenten, die Netzwerkgeräte verwalten. Die Künstliche Intelligenz kennt intelligente Agenten. Im Internet ist es derzeit Mode, alle Programme Agenten zu nennen, die auf einer Webseite lokal beim Benutzer etwas berechnen.

Im Prinzip haben auch alle Autoren recht. Ein Agent ist "jemand, der im Auftrag jemandes anderen eine Aufgabe erledigt" [Webster92]. Diese Definition deckt zwar das Spektrum der Agentenanwendungen ab, ist aber zu allgemein. Wir beschränken die Definition deshalb auf den Informatikbereich. Ein Agent ist demzufolge ein Programm, das während seiner Ausführung im Auftrag und zum Nutzen eines anderen Programms oder Menschen eine Aufgabe ausführt.

Nachdem wir definiert haben, was ein Agent ist, betrachten wir nun spezielle Agenten, die in dieser Diplomarbeit eine besondere Rolle spielen: Die mobilen Agenten.

I.3.2. Der mobile Agent

Mobile Agenten erledigen allein oder in Kooperation mit anderen Agenten eine Aufgabe. Zur Erfüllung ihrer Aufgabe haben sie die Möglichkeit, autonom von Rechner zu Rechner zu migrieren. Das bedeutet, daß sich ein mobiler Agent während des Programmablaufs entscheiden kann, auf einen anderen Rechner zu wechseln und dort weiterzuarbeiten. Zur Durchführung der Migration wird der aktuelle Status des Agenten verpackt, zum Zielrechner geschickt und dort wieder ausgepackt. Nach der Migration arbeitet der mobile Agent an der Stelle im Programmcode weiter, an der er auf dem vorigen Rechner aufgehört und beschlossen hat, zu migrieren.

Mobile Agenten beschreiben also ein neues Programmierparadigma für verteilte Systeme. Diese Beobachtung ist sehr wichtig, denn sie hat gewaltige Auswirkungen auf Anwendungsszenarien mit mobilen Agenten. Es gibt de facto keine Anwendung, die sich nicht auch ohne mobile Agenten konstruieren ließe. Diese Situation ist ähnlich derjenigen Anfang der 80er Jahre, als die objektorientierte Programmierung aufkam. Niemand konnte ein wirklich gutes Beispiel zugunsten objektorientierter Programmierung vorlegen, das sich nicht auch prozedural lösen ließ. Die einzigen immer wieder genannten Vorteile waren bessere Wartbarkeit des Programmcodes und leichtere Modellierung der Anwendungslogik. Ähnlich sieht es heute mit mobilen Agenten aus. Es ist ein schönes Programmierparadigma, mit dem sich die Komplexität von verteilten Systemen vielleicht besser handhaben läßt. Als

Beispiele stellen wir in Kapitel I.4 Szenarien vor, in denen die Modellierung der Anwendungslogik unter Verwendung von mobilen Agenten einfacher oder logischer als mit konventionellen Verfahren ist. In [Harrison95] werden weitere Szenarien vorgestellt und bewertet. Darüberhinaus findet sich dort eine Analyse des Agentenparadigmas.

Neben dem besseren Programmierparadigma haben mobile Agenten noch andere vorteilhafte Eigenschaften, die sie für manche Einsatzzwecke prädestinieren.

- **Erweiterung von Servercode**
  Durch die Fähigkeit zu einem Server zu migrieren und dort lokal Programmcode auszuführen, wird effektiv die Funktionalität des Servers erweitert. Der Agent kommuniziert dazu über eine Schnittstelle mit dem Server, der den Agenten mit den gewünschten Informationen versorgt oder über Zustandsänderungen informiert.

- **Reduzierung von Kommunikationskosten**
  Durch die lokale Ausführung von Entscheidungen lassen sich, insbesondere bei Anwendungen, die Informationen sammeln, zeitaufwendige und netzbelastende Langstreckenkommunikationen einsparen.

- **vereinfachte Programmierung von parallelen Anwendungen**
  Das Konzept der mobilen Agenten ist bereits im Grundansatz hochgradig parallel. Zur Durchführung einer Aufgabe, die zwei parallele Tasks benötigt, werden einfach zwei Agenten erzeugt, von denen jeder eine Teilaufgabe ausführt.

- **Mobilität des Anwenders**
  Es gibt in letzter Zeit starke Bestrebungen PDAs marktreif zu machen. Diese können per Funk oder über eine normale Telefonverbindung in bestehende Computernetzwerke eingebunden werden. Agenten, die von PDAs ins Netz geschickt werden, können ihre Aufgabe ausführen, ohne daß während der ganzen Ausführungszeit eine Netzverbindung bestehen muß. Der Anwender ist nicht an einen festen Arbeitsplatz gebunden, sondern kann die Agenten losschicken und bei der nächsten Netzverbindung kehren sie mit den Ergebnissen der Anfragen zurück.

- **flexiblere Migration**
  Im Gegensatz zu Remote Execution können mobile Agenten zu einem beliebigen Zeitpunkt entscheiden auf einen anderen Rechner zu wechseln. Dies kann auch mitten in einer Berechnung sinnvoll sein, zum Beispiel für eine bessere Lastbalancierung. Ob diese Autonomität noch mehr Vorteile hat, konnte bisher noch nicht gezeigt werden [Baumann97b].

I.3.3. Agentensysteme

Agenten migrieren zwischen verschiedenen Rechnern weltweit hin und her und führen dort auch noch Programmcode aus. Jedem Systembetreuer sträuben sich bei diesem Gedanken die Haare. Das klingt doch alles sehr nach Viren. Das Sicherheitsbedürfnis der Rechner, die Agenten beherbergen (Hosts), ist ohne Zweifel berechtigt. Hinzu kommt weitere Funktio-

nalität, die von Agenten benötigt wird: Der Agentenstatus muß für die Migration eingepackt, verschickt und wieder ausgepackt werden ("Marshalling" / "Unmarshalling"). Eventuell muß Code für den Agenten nachgeladen werden, wenn er auf dem neuen Rechner bisher nicht bekannt ist. Die Kommunikation mit andern Agenten lokal und entfernt muß organisiert werden. Zudem muß der Zugriff auf Rechnerressourcen koordiniert und überwacht werden.

Diese Aufgaben werden von einer speziellen Instanz auf dem betroffenen Rechner erledigt, dem Agentensystem. Das Agentensystem bildet eine Hülle um die Agenten. Agenten benötigen die Einrichtungen und die Unterstützung des Agentensystems, um auf einem fremden Rechner aktiv werden zu können. Ein Agent hat keine Chance, unkontrolliert aus dieser Hülle auszubrechen und im Rechner "herumzuspuken". Das Agentensystem definiert sogenannte "Orte", an denen sich Agenten aufhalten können und die jeweils besondere Dienste anbieten können.

Der Hostrechner ist also vor den Agenten in Sicherheit. Anders sieht das für die Agenten aus. Da das Agentensystem die volle Kontrolle über die Agenten hat, muß es vertrauenswürdig sein. Der Schutz der Agenten vor nichtvertrauenswürdigen Agentensystemen ist ein ernsthaftes Problem. Es ist aber fraglich, ob sich sinnvolle Lösungen dafür finden lassen.

I.3.4. MOLE

MOLE ist ein Agentensystem, das am IPVR der Universität Stuttgart entwickelt wird. Es basiert auf der Diplomarbeit von Fritz Hohl [Hohl95] und bietet eine funktionierende Umgebung zur Erforschung des Agentenparadigmas. Die Betonung liegt hierbei auf funktionierend und Forschung. Es gibt weltweit nur sehr wenig Agentensysteme, deren Komplexität den Grad von MOLE erreicht. Andererseits müssen bis zu einer kommerziellen Einsatzfähigkeit des Systems noch viele Fehler gefunden, Sicherheitslöcher gestopft und Designentscheidungen getroffen werden.

Sowohl das Agentensystem MOLE, als auch die Agenten selber, sind vollständig in Java geschrieben. Eine ausführliche Betrachtung der Vor- und Nachteile von Java als Sprache für Agenten und das System findet sich in [Hohl95].

Orte werden in MOLE "Locations" genannt. Eine "Engine" ist von außen betrachtet nichts weiter als ein UNIX-Prozeß, in dem ein Java-Interpreter läuft. Sie verwaltet einen oder mehrere Orte und bietet die grundlegene Infrastruktur eines Agentensystems. Darunter fallen Threadverwaltung, Codeserver und Kommunikationssubsysteme. Agenten sind immer an eine Location gebunden, die einen eindeutigen Namen besitzt. Die Struktur der Namen hält sich an das Namensschema des Domain Name Systems (DNS), das, je nach Konfiguration der Engine, zur Namensauflösung von Orten verwendet wird. Agenten kennen nur die Namen von Orten. Es ist irrelevant auf welchem Rechner oder in welcher Engine sich der Ort befindet.

Als Schnittstelle zwischen dem Rechner und den mobilen Agenten fungieren in MOLE die Systemagenten. Sie gehören logisch zum Agentensystem und sind nicht mobil. System-

agenten haben im Vergleich zu mobilen Agenten mehr Rechte und dürfen beispielsweise auf das lokale Dateisystem des Hostrechners zugreifen. Mit Systemagenten, die bestimmte Dienste anbieten, lassen sich beliebige Schnittstellen zu Programmen außerhalb des Agentensystems realisieren.

I.3.5. Agentengruppen

Agenten agieren im allgemeinen nicht ohne Kommunikation mit anderen Agenten. Im Gegenteil, sie erfüllen ihre Aufgaben im Verbund mit anderen Agenten. Als Beschreibung für diesen Umstand bietet sich der Begriff der "Agentengruppe" an.

Agentengruppen bestehen aus den beteiligten Agenten, einem privaten Kommunikationskanal, an dem nur Agenten dieser Gruppe teilnehmen können, sowie einem "group coordinator".

Der Group Coordinator (GC)

Der GC hat zwei Aufgaben. Er repräsentiert die Gruppe nach außen hin und kann selbst wieder Mitglied in anderen Gruppen sein ("external group coordinator"). Auf diese Weise lassen sich Hierarchien von Agentengruppen erzeugen. Sinnvoll ist eine verteilte Implementierung des GC um einen "Single Point of Failure" (SPOF) zu vermeiden. Der GC überwacht die Ausführung der Aufgabe der Gruppe und bestimmt, wann die Aufgabe erfüllt ist ("internal group coordinator").

In der Funktionalität ähnelt der GC dem Modell der Actors [Agha87]. Actors erzeugen als Reaktion auf eingegangene Events neue Events und modellieren dadurch die Anwendungslogik. Das Konzept des GCs geht noch etwas über dieses Modell hinaus indem Timeouts eingeführt werden. So erhält man die Möglichkeit auf unterbrochene Netzverbindungen oder Prozesse, die nicht mehr antworten, zu reagieren.

Die Gruppenbedingung

Die Bedingung für die Erfüllung einer Aufgabe kann sehr einfach sein:

- UND-Gruppe: Alle Agenten müssen ihre Teilaufgabe erfüllen.
- ODER-Gruppe: Ein Agent muß seine Teilaufgabe erfüllen.

Die Bedingung darf aber auch beliebig komplex werden, wobei hier noch zu untersuchen bleibt, wie komplex Bedingungen sein müssen, um "vernünftige" Aufgaben zu beschreiben. Auf Basis der in der Gruppe versendeten Events kann der GC den aktuellen Status der Gruppe bestimmen [Baumann97b].

Gruppensemantiken

Die bisher beschriebenen Gruppen nennen wir Terminierungsgruppen, da die Zugehörigkeit eines Agenten zu der Gruppe durch seine Lebensaufgabe definiert ist. Die Lebensdauer der Gruppe definiert sich aus der Tatsache, ob die im internal GC verwaltete Bedingung zu wahr evaluiert. Ist dies der Fall, so hat die Gruppe ihre Aufgabe erfüllt und kann terminiert werden. Terminierungsgruppen sind eine Möglichkeit, um Terminierung und Waisenerkennung von Agenten durchzuführen (siehe dazu die Kapitel II.3 und II.4). Natürlich kann ein Agent gleichzeitig nur einer Terminierungsgruppe angehören.

Andere Gruppensemantiken sind denkbar. Szasz betrachtet in seiner Studienarbeit beispielsweise Synchronisierungsgruppen [Szasz97].

I.4. Agentenszenarien

Dieses Kapitel beschreibt Anwendungsszenarien, in denen Agenten eingesetzt werden können. Es dient zum einen der Einführung in die Arbeit mit mobilen Agenten und liefert uns zum anderen Anwendungsfälle, aus denen wir Anforderungen an einen Eventmanager ableiten können. Weitere Szenarien finden sich in [Harrison95].

I.4.1. Reiseplanung

Ein Mensch will verreisen. Heute geht er in ein Reisebüro oder wühlt in den dicken Katalogen der Reiseveranstalter. Das geht aber auch mit Agenten: Der Reisewillige gibt die Eckdaten seiner Reise (möglicher Zeitraum, Ausgangsort, Zielort oder -region, Hotelkategorie, maximale Kosten, ...). Die Applikation erzeugt daraufhin Agenten, von denen sich jeder um einen Teilaspekt der Aufgabe kümmert. Einer prüft Flugmöglichkeiten, ein anderer versucht es bei der Bahn, ein dritter klappert Hotels ab, der nächste sucht ein Mietauto falls kein eigener PKW mitgenommen werden soll, ... Dabei kommunizieren die Agenten jeweils über Events und tauschen die neuesten Ergebnisse aus. Eventuelle Rückfragen an den Benutzer werden über die Applikation ausgewertet. Die Konzeption der Reise kann nach unterschiedlichen Kriterien gewichtet und bewertet werden. Ist ein Optimum gefunden oder alle sinnvollen Kombinationen durchprobiert, kehren die Agenten zum Auftraggeber zurück oder werden terminiert.

Anforderungen

- verhältnismäßig kleine Gruppe (max. 10-15 Agenten)
- hochmobil (Agenten migrieren von Anbieter zu Anbieter)
- viel Eventkommunikation (Austausch der besseren Suchergebnisse)
- "intelligente" Agenten notwendig

I.4.2. WWW-Recherche

Ein Anwender wählt sich kurzzeitig ins Internet von seinem Laptop aus ein und schickt einen Agenten mit einer Anfrage los. Dieser erzeugt mehrere Kindagenten, die zu unterschiedlichen Suchindizes im World Wide Web migrieren und dort parallel Anfragen starten, Ergebnisse bewerten und als Reaktion unter Umständen genauere oder umfassendere Anfragen stellen. Die Kommunikation der Agenten untereinander erfolgt wieder mit Events, wobei ein Rating erstellt wird, wie gut das Ergebnis der Anfrage des Anwenders eingeschätzt wird. Beim nächsten Netzconnect erhält der Anwender das Rating und auf Wunsch die Ergebnisse.

Anforderungen

- kleine Gruppe von Agenten
- wenig mobil
- Kommunikationsaufwand in Gruppe nicht genau einzuschätzen
- Anwender und Applikation nicht ständig verfügbar

I.4.3. Lokale Beobachter

Es gibt viele Nachrichtenquellen im Netz, die von unterschiedlichen Leuten unterschiedlich ausgewertet werden. Agenten können von ihrem Auftraggeber direkt zur Nachrichtenquelle geschickt werden, dort auf die wirklich relevanten Daten warten und dann je nach Programmierung autonom auf die Nachricht reagieren. Beispiele dafür sind Agenten, die Maschinen überwachen (sehr harte Echtzeitanforderungen!), Börsenkurse untersuchen und statistisch auswerten (unter Umständen hoher Rechenaufwand), Wetterdaten verarbeiten (riesige Datenmengen), oder für Autofahrer interessant: Agenten, die direkt in der Unfall- oder Staumeldestelle sitzen und immer wissen, wo ihr Besitzer gerade hinfährt und ihn so vor Unfällen oder Staus warnen können.

Anforderungen

- möglicherweise sehr viele Agenten sehr weit verteilt
- kaum Migration
- wenig Eventkommunikation
- unter Umständen hoher Rechenaufwand auf Host (Wetter! Börse!)
- eventuell hohe Anforderungen an Ausfallsicherheit (Replikate, Antwortzeit)

I.4.4. Literaturrecherche im Internet

Wissenschaftliche Publikationen werden heute schon sowohl in gedruckter, als auch in elektronischer Form publiziert. Aus Gründen der Vereinfachung werden viele Publikationen als komprimierte Postscriptdateien abgelegt. Bald werden so gut wie alle aktuellen Publikationen elektronisch verfügbar sein. Intelligente Agenten bieten sich daher zur Literaturrecherche geradezu an. Der Anwender übergibt einem Rechercheagenten eine Referenz auf ein Dokument, sowie Kriterien, anhand derer interessante Dokumente vom Rechercheagenten erkannt werden können. Dieser untersucht die im Dokument vorhanden Literaturangaben und schickt seinerseits Rechercheangenten los, die für die entsprechenden Literaturstellen zuständig sind. Die Rechercheagenten können lokal auf einem Server die Referenzen abarbeiten, Dokumente dekomprimieren, durchlesen und neue Agenten mit den Referenzen losschicken. Auf diese Weise wird im Extremfall die vollständige transitive Hülle über alle referenzierten Dokumente gebildet. Mit Eventkommunikation zwischen den Agenten untereinander und mit dem Benutzer können Ringschlüsse erkannt und der Suchraum sinnvoll eingeschränkt werden.

Anforderungen

- viele Agenten weit verteilt
- unter Umständen starke Eventkommunikation
- viele Migrationen

I.4.5. Elektronische Märkte

In diesem Bereich machten vor allem die Whitepapers von General Magic [White94] Furore. Agenten werden vom Anwender mit den Daten eines bestimmten Produkts auf die Suche geschickt. Beispielsweise eines bestimmten Fotoapparats eines bestimmten Herstellers und einem maximal zu zahlendem Preis. Der Agent migriert zunächst zu einem Ort an dem elektronische "Gelbe Seiten" verfügbar sind, sucht dort alle Kamerahändler heraus und migriert dann auf dem elektronischen Marktplatz von Händler zu Händler um das beste Angebot zu suchen (oder auch das billigste). Mit den besten drei Ergebnissen der Suche kehrt er zum Anwender zurück, dieser wählt aus, wo er kaufen möchte. Der Agent migriert zu diesem Händler und bestellt die Kamera, womöglich wird auch gleich elektronisch gezahlt und die Kamera kurze Zeit darauf geliefert.

Für MOLE gibt es in diesem Bereich eine prototypische Implementierung von Klaus Villinger [Villinger96].

Anforderungen

- hohe Sicherheitsstandards für Agenten und Server (es geht hier um echtes Geld!)
- unter Umständen viele Migrationen
- Protokolle zur Aushandlung der Parameter bei "Beratung" des suchenden Agenten ("Verkaufsgespräch") nötig

I.4.6. Folgerungen

Agenten können in vielen extrem unterschiedlichen Szenarien eingesetzt werden. Einige Gemeinsamkeiten treten aber in fast allen Szenarien auf. Meist schickt ein Anwender einen Agenten los, der zur Erledigung seiner Aufgabe einen oder mehrere Kindagenten erzeugt. An einem Ort entsteht also eine Agentengruppe, die mit der Zeit wachsen kann und sich im Netz ausbreitet. Die Größe der Gruppen ist je nach Szenario sehr unterschiedlich. Allgemein läßt sich sagen, daß Gruppen mit sehr aktiven, oft migrierenden Agenten eher klein sind, während Gruppen von Agenten, die Überwachungsaufgaben ausführen, sehr viele Teilnehmer haben können, die dafür im Gegenzug selten migrieren.

Die Kommunikation innerhalb der Gruppe kann in beiden Fällen sehr stark sein. Da der Eventmanager für sehr stark migrierende Gruppen andere Mechanismen verwenden kann, als für eher statische Gruppen, ist es sinnvoll, ihm soviel Informationen wie möglich über die Gruppe zur Verfügung zu stellen. Damit befassen wir uns im zweiten Teil dieser Arbeit.

II. Anforderungen an das Eventmanagement

Anforderungen an einen Eventmanager können nur auf der Basis der verwendenden Anwendungen bestimmt werden. Die Anwendung spezifiziert, welche Eigenschaften und Garantien der Eventmanager bieten kann und muß.

In diesem Teil der Diplomarbeit werden zunächst grundsätzliche Anforderungen entwickelt, die sich direkt aus dem Management von Events ergeben. Danach werden Anforderungen untersucht, die sich aus den Anwendungsbereichen Terminierung, Waisenerkennung und Synchronisation, sowie aus den im ersten Teil vorgestellten Szenarien ergeben. Es werden Aussagen zu Eigenschaften von Events und zum Verhalten im Fehlerfall gemacht.

II.1. Grundlegende Anforderungen

Die folgenden Anforderungen können für jede Anwendung des Eventmanagements vorausgesetzt werden.

II.1.1. Verteilung

Wie aus der Aufgabenstellung hervorgeht, darf der Eventmanager keinen "Single Point Of Failure" haben. Der Mechanismus des Eventmanagements darf nicht auf der Erreichbarkeit und dem Funktionieren einer zentralen Einheit oder eines zentralen Servers beruhen.

Dies hat hauptsächlich zwei Gründe. Zum einen ist der Kommunikationsaufwand mit der Zentrale in einem weit verteilten System unter Umständen erheblich und zum anderen bricht das komplette Eventmanagement-System zusammen, wenn ein einziger Rechner oder womöglich nur ein einzelner Prozeß ausfällt.

Andererseits kann es unter Umständen vorteilhaft sein, einen ausgezeichneten Knoten zu bestimmen, der zentrale Aufgaben ausführt (zum Beispiel Vergabe eindeutiger globaler Nummern für Events eines Eventchannels). Dies kann man zwar auch verteilt lösen, allerdings steigt dann der Kommunikationsaufwand eventuell erheblich an. Es muß im Einzelfall sorgfältig geprüft werden, welche Lösung für den gewünschten Zweck besser geeignet ist. Lösungen, die solche ausgezeichneten Knoten verwenden, müssen Mechanismen zur Verfügung stellen, die im Falle eines Fehlers (Ausfall der Zentrale, Netzpartitionierung, Absturz) die Funktionsfähigkeit des Eventmanagers weiterhin gewährleisten.

II.1.2. Die Schnittstelle zum Eventmanager

An der Schnittstelle zwischen Anwendung und Eventmanager erwacht ein Event zum Leben. Er wird von einem Supplier an den Eventmanager übergeben, beziehungsweise von einem Consumer empfangen. Die Verwendung der Schnittstelle soll von Anwendungsseite aus möglichst einfach sein, damit sich Programmierer einer Anwendung nicht erst mit einer komplexen Anbindung an den Eventmanager beschäftigen müssen, sondern sich möglichst schnell ihrer eigentlichen Aufgabe zuwenden und den Eventmanager als Werkzeug verwenden können.

Prinzipiell werden folgende Funktionen in der Schnittstelle benötigt.

- **Definition von Eventtypen**
  Bevor der Eventmanager Events eines bestimmten Typs verteilen kann, muß er diesen Eventtyp zunächst kennenlernen.

- **Registrieren der Consumer**
  Objekte der Anwendung können ihr Interesse an spezifizierten Eventtypen anmelden.

Der Eventmanager sorgt dafür, daß die Objekte ab diesem Zeitpunkt Events erhalten. Wenn der Eventmanager unterschiedliche Dienstkategorien anbietet, soll bei Anmelden des Interesses vermerkt werden, welche Kategorie verwendet werden soll.

- **Senden von Events durch Supplier**
  Ein Supplier soll auf einfache Art und Weise Events eines spezifizierten Typs in das System einbringen können.

- **Empfangen von Events durch Consumer**
  Ein Consumer soll auf einfache Art und Weise Events vom System erhalten können. Dies kann wahlweise synchron auf Anforderung des Consumers oder asynchron bei Eintreffen eines Events geschehen.

Der Eventmanager ist für die Anwendung weitestgehend eine "Black-Box". Die Anwendung muß von den spezifizierten und bestätigten Dienstmerkmalen ausgehen können. Darunter fallen zum Beispiel die Fehlersicherheit, die Verfügbarkeit eines "Eventarchivs" und ähnliche Dinge. Es sollte für die Anwendung trotzdem möglich sein, Informationen über den Status des Systems zu erhalten, zum Beispiel indem auf Anforderung sogenannte Systemevents, die den Status des Systems beschreiben, an die Anwendung gesendet werden.

Ebenso sollen die Objekte, die den Eventmanager nutzen, für den Eventmanager eine Black-Box sein. Die Schnittstelle muß so spezifiziert sein, daß sie von einem beliebigen Objekt implementiert werden kann. Unabhängig davon, zu welcher Klasse das Objekt gehört.

II.1.3. Eventchannels

In einem großen Eventmanagementsystem werden sehr viele Events versendet. Allerdings interessiert nicht jeder Event eines bestimmten Typs oder Inhalts jeden Consumer. Im Gegenteil, es ist eher zu erwarten, daß es Gruppen von Consumern und Suppliern gibt, die eine gemeinsame Semantik besitzen. Innerhalb zweier verschiedener Gruppen werden möglicherweise Events desselben Typs versendet, deren Inhalt aber die Mitglieder der jeweils anderen Gruppe nicht interessiert und die daher dort vom Eventmanager nicht ausgeliefert werden müssen. Eine solche Ansammlung von Supplier- und Consumerobjekten unter einer eindeutigen Bezeichnung soll im folgenden als Objektgruppe bezeichnet werden. Innerhalb einer Objektgruppe wird über einen Eventchannel kommuniziert. Die in der Einführung beschriebenen Agentengruppen entsprechen diesem Modell.

Der Eventchannel wird in diesem einfachen Modell (im Gegensatz zum OMG-Modell, Kapitel III.1) lediglich als Kommunikationsmittel verwendet. Der Eventchannel verbindet die einzelnen Objekte miteinander. Events, die vom demselben Supplier in den Channel gesendet werden, kommen bei jedem Consumer in der Sende-Reihenfolge an. Welche weiteren Garantien der Eventchannel bieten muß, wird in den folgenden Kapiteln untersucht.

Das Konzept des Eventchannels erlaubt es dem Eventmanager, die interne Organisation möglichst effizient zu gestalten, indem ein Event tatsächlich nur an diejenigen Consumer verteilt wird, die sich auch wirklich dafür interessieren. Andere Consumer sehen diese Events nicht, und die Kommunikationskosten können auf das unbedingt notwendige Minimum gesenkt werden. Alle Netzknoten, die an dem Eventchannel teilnehmen, bilden die Knotengruppe des Channels. Es genügt, wenn der Eventchannel nur auf den Knoten der Knotengruppe bekannt ist, solange es Mechanismen gibt, einen Eventchannel zu finden, der auf dem aktuellen Knoten unbekannt ist.

Auf welche Art und Weise Gruppen und Channels realisiert werden, bleibt dem Eventmanager überlassen. Denkbar ist, tatsächlich jede Gruppe mit einem eigenem Channel auszustatten oder alle Events aller Gruppen über eine gemeinsame Kommunikationsstruktur zu verschicken und für jede Gruppe die relevanten Events entsprechend auszufiltern.

II.1.4. Authentifikation und Autorisierung

Für einige Anwendungen (zum Beispiel Terminierung) kann es sinnvoll sein, den Zugriff auf einen Eventchannel zu beschränken und nur bestimmten Objekten (zum Beispiel Locations) den Zugriff zu erlauben und anderen (zum Beispiel Agenten) nicht. Der Eventmanager soll nur mit Objektidentifikatoren arbeitten, um möglichst universell zu bleiben. Daher sind für ihn Authentifikation und Autorisierung für den Zugriff auf Eventchannels aufwendiger zu realisieren als für das Agentensystem. Die Beschränkung des Zugriffs von Agenten auf bestimmte Eventchannels ist innerhalb des Agentensystems einfacher zu realisieren.

Betrachten wir die Agentengruppen aus der Einführung. Anstatt die Agenten direkt den Eventchannel spezifizieren zu lassen, arbeiten sie nur mit einer Referenz auf den Channel. Jede Aktion mit einem Eventchannel, die ein Agent durchführen darf, benötigt die Channelreferenz. Nur beim Anlegen des Channels wird dieser explizit im Eventmanager benannt. Als Ergebnis erhält der Benutzer des Eventchannels eine Referenz darauf und kann diese nun an beliebige Agenten weitergeben. Wenn ein Objekt, das natürlich selbst ein Agent sein kann, eine Agentengruppe anlegt und nur den Agenten und dem Coordinator der Gruppe die Referenz auf den Eventchannel mitteilt, so ist dieser Eventchannel quasi privat für die Gruppe, da kein anderer Agent die Referenz des entsprechenden Eventchannels kennt. Wenn für die Agenten keine Möglichkeit zur Verfügung gestellt wird, existierende Eventchannels direkt beim Eventmanager abzufragen beziehungsweise vom Agentensystem diese Informationen so gefiltert werden, daß keine privaten Eventchannels des Agentensystems ausgegeben werden, so sind aus Sicht der Agenten private Kanäle möglich.

Daraus folgt, daß der Eventmanager im Rahmen dieser Arbeit keine Sperrmechanismen für den Zugriff auf bestimmte Eventchannels anzubieten braucht. Die Funktionalität muß vom Agentensystem erbracht werden. Wie oben dargelegt, ist das aber verhältnismäßig einfach und ohne großen Aufwand möglich.

II.1.5. Weitere Anforderungen

In diesem Abschnitt stellen wir weitere Anforderungen an den Eventmanager.

- **Effizienz**
  Der Eventmanager soll die Kommunikation zwischen Suppliern und Consumern mit möglichst geringen Kosten bewerkstelligen. Gleichzeitig soll die Verzögerung zwischen dem Senden eines Events und der Auslieferung des Events an alle Consumer möglichst kurz sein.

- **Ortstransparenz**
  Ein Event, der von einem Supplier an den Eventmanager übergeben wird, soll an jeden interessierten Consumer ausgeliefert werden, egal wo sich der Consumer befindet. Wenn ein Consumer nicht erreichbar ist, muß angegeben werden können, was mit dem Event in diesem Fall geschehen soll.

- **Mobilität**
  Sowohl Supplier, als auch Consumer dürfen mobil sein. Das bedeutet, daß sie von einem Rechner zu einem anderen migrieren können, ohne daß sie die Verbindung zum Eventmanager verlieren. Der Eventmanager muß sicherstellen, daß migrierende Objekte die gleichen Garantien in Hinblick auf Quality-of-Service erhalten, wie stationäre Objekte.

- **Anonymität**
  Dies hängt indirekt mit der Ortstransparenz zusammen. Ein Supplier sendet einen Event ohne zu wissen, welche Objekte sich dafür interessieren. Der Supplier muß die Empfänger des Events nicht kennen und braucht sie nicht zu kennen. Stattdessen wird über einen oder mehrere Eventchannels kommuniziert, deren Bezeichnungen Supplier und Consumer bekannt sind. Nur der Eventmanager hat den Überblick darüber, an welchen Orten Teilnehmer des Channels sitzen.

II.2. Charakteristik von Eventchannels

In den später vorgestellten Protokollen werden Eventchannels verwendet. Für den Durchsatz und die Auslegung eines Eventmanagers ist es vorteilhaft zu wissen, wie solche Eventchannels aussehen. Ob sie viele oder wenige Mitglieder haben oder wie sie geographisch verteilt sind. Mit diesen Kriterien können Eventmanager geprüft oder auf besonders typische Fälle eines Protokolls hin optimiert werden. Falls ein Eventmanager verschiedene Protokolle zur Verfügung stellt, ist es für den Programmierer einer Anwendung möglich, je nach gewünschter Funktionalität das passende Protokoll zu wählen.

II.2.1. Kriterien

In den folgenden Abschnitten werden Kriterien für die Einordnung von Knotengruppen im obigen Sinne erarbeitet und zum Teil formal beschrieben. Anhand dieser Kriterien können Anwendungen, die Eventchannels verwenden, klassifiziert und die Auswirkungen von Protokollen des Eventmanagers abgeschätzt werden. Damit ist es möglich für einen konkreten Anwendungsfall ein passendes Eventmanagementprotokoll zu konstruieren, indem die Parameter des Anwendungsfalls klassifiziert werden und das Protokoll entsprechend angepaßt wird.

Das logische Konstrukt Eventchannel betrachten wir hier mehr aus Sicht des Netzwerks. Ein Knoten ist Mitglied der Knotengruppe eines Eventchannels, wenn sich auf dem Knoten mindestens ein Consumer des Eventchannels befindet. Es ist unerheblich, ob dies im Falle von MOLE Engines, Locations oder Agenten sind. All diese Objekte müssen sich über ihren lokalen Eventmanager beim Eventdämon des Knotens als Consumer anmelden. Pro Knoten gibt es damit genau einen Ansprechpartner des Eventmanagers, der stellvertretend für alle beteiligten Objekte auf diesem Knoten agiert. Ein Event, der in den Eventchannel gesendet wird, muß innerhalb der Knotengruppe verteilt werden.

Einige Kriterien lassen sich besser aus Sicht des Netzwerks, andere besser aus Sicht der Agenten beschreiben.

Gruppengröße

Unter Gruppengröße verstehen wir die Anzahl von Knoten oder Agenten, die an der Gruppe beteiligt sind. Je größer die Gruppe ist, desto wahrscheinlicher ist ein Knoten des Netzwerks an dieser Gruppe beteiligt, und desto mehr lohnen sich Verfahren einzelne Knoten oder Unterbäume eines spannenden Baums im Eventmanager gemeinsam zu behandeln.

Die Gruppengröße im Verhältnis zu der Größe des Netzwerks (der Anzahl Knoten im Netzwerk) ist ein wichtiges Maß zur Feststellung, welcher Algorithmus effizient angewendet werden kann. In [Belkeir89] beschreiben Belkeir und Ahamad die verblüffende Eigenschaft von Walls Algorithmus [Wall80], der einen spannenden Baum bei jeder Änderung der Gruppe neu aufbaut, daß er bei kleinen Gruppengrößen anderen Algorithmen, die

auf einen Neuaufbau verzichten und stattdessen Heuristiken zum Verändern des Baums anwenden, zumindest ebenbürtig ist. Wir werden dies später im Detail diskutieren.

Die relative Gruppengröße läßt sich formal als der Anteil der Teilnehmerknoten des Eventchannels im Verhältnis zur Gesamtzahl aller am Eventmanagement beteiligten Knoten des Netzwerks beschreiben:

$$Größe_{KnGruppe} = \frac{\#Knoten_{Gruppe}}{\#Knoten_{gesamt}} \qquad (II.1)$$

mit

$\#Knoten_{Gruppe}$ = Anzahl Knoten, die Teilnehmer des Channels sind,
$\#Knoten_{gesamt}$ = Anzahl aller Knoten.

Die relative Gruppengröße von Knotengruppen läßt sich rechnerisch verhältnismäßig einfach ermitteln. Insbesondere die Gesamtzahl von Knoten ist ein vergleichsweise statischer Wert. Die Formel läßt sich natürlich immer nur in einem konkreten Einzelfall anwenden.

Schwieriger, und um einiges ungenauer, ist es, entsprechende Werte für die relative Größe von Agentengruppen zu bestimmen, da die Gesamtzahl der Agenten im System sehr starken Schwankungen unterliegen kann. Eine Bestimmung der Anzahl aller Agenten im System ist in Wirklichkeit nur eine grobe, ungenaue Abschätzung, die zudem nur im Augenblick der Messung gültig ist. Eine solche Abschätzung bringt für den Eventmanager keine neuen Informationen, da nur die Verteilung der Events zwischen den Knoten einer Gruppe einen Einfluß auf den Durchsatz hat und näherer Untersuchungen bedarf.

Gruppenanzahl

Mit diesem Maß beschreiben wir die Anzahl der gleichzeitig aktiven Gruppen, bezogen auf die Gesamtzahl der Knoten. Je mehr Gruppen im Netzwerk vorhanden sind, desto mehr Informationen müssen vom Eventmanager verwaltet werden. Ein Knoten, der Mitglied vieler Gruppen ist, muß für jede Gruppe die Verwaltungsinformation für den zugehörigen spannenden Baum speichern. Dies kann starke Auswirkungen auf den Speicherplatzverbrauch und den Durchsatz des Knotens haben.

$$Zahl_{Gruppen,\ gesamt} = \frac{\#Gruppen}{\#Knoten_{gesamt}} \qquad (II.2)$$

$Zahl_{Gruppen,\ gesamt}$ ist um so höher, je mehr Gruppen im Netzwerk vorhanden sind. Sind keine Gruppen aktiv, so ist $Zahl_{Gruppen,\ gesamt} = 0$. Ist jeder Knoten Mitglied genau einer Gruppe, so ist $Zahl_{Gruppen,\ gesamt} = 1$. Je höher der Wert ist, desto mehr Gruppen müssen im Schnitt pro Knoten verwaltet werden und desto mehr lohnen sich Verfahren, die die Behandlung von Gruppen auf den Knoten vereinfachen oder beschleunigen.

Gruppenausdehnung

Darunter verstehen wir die geographische Verteilung einer Gruppe, daß heißt wie sich die Teilnehmerknoten einer Gruppe über das vorhandene Netzwerk verteilen. Dies hat Auswirkungen auf die Implementierung und die Effizienz der globalen Kommunikation zwischen den betroffenen Knoten. Gegenüber einem LAN steigt bei Langstreckenkommunikation zum einen die Verzögerung durch die Entfernung deutlich an. Zum anderen sinkt, wegen der meist geringeren Bandbreiten, der Durchsatz deutlich ab. Dies hat direkte Auswirkungen auf den Overhead, den ein Protokoll erzeugen darf. Je mehr Overhead, desto teurer ist der einzelne Event.

$$Ausdehnung_{Gruppe} = \frac{Kommunikationskosten_{Gruppe}}{Kommunikationskosten_{gesamt}} \qquad \text{(allgemein)} \qquad (II.3)$$

$$= \frac{\sum_{\text{Kanten E von MSB}_{\text{Gruppe}}} W(E)}{\sum_{\text{Kanten E von MSB}_{\text{gesamt}}} W(E)} \qquad \text{(spannender Baum)} \qquad (II.4)$$

Der Wert von $Ausdehnung_{Gruppe}$ errechnet sich allgemein aus den Kommunikationskosten für die Verteilung eines Events in der Gruppe geteilt durch die Kommunikationskosten für die Verteilung des gleichen Events zu allen Knoten *(II.3)*. Für minimal spannende Bäume errechnet sich der Wert aus der Summe der Gewichte der Kanten des minimal spannenden Baums über alle Knoten der Gruppe geteilt durch die Summe der Gewichte der Kanten des minimal spannenden Baums über alle Knoten der Gruppe *(II.4)*.

> *MSB_{Gruppe} darf auch Knoten enthalten, die nicht Teil der Gruppe sind, wenn die Kommunikationskosten im Baum dadurch verringert werden.*

Zu beachten ist, daß sich die Kommunikationskosten zwischen den Knoten laufend ändern können. Der Wert von $Ausdehung_{Gruppe}$ stellt wieder einmal nur einen Schnappschuß des Zustands zum Zeitpunkt der Messung dar.

Die Formel *(II.4)* berücksichtigt nur das "Gewicht" von Kanten des Gruppen-MSBs und errechnet so den Anteil des Gewichts der Gruppe an dem Gewicht des Gesamt-MSBs. Die zugegebenermaßen künstliche Angabe "Gewicht einer Kante" wird auch zur Berechnung des minimal spannenden Baums verwendet. Zur Berücksichtigung der oben erwähnten Charakteristika von Leitungen zwischen den einzelnen Knoten können diese gewichtet in die Berechnung eingehen:

$$W(E) = a \times delay(E) + \frac{b}{capacity(E)} \qquad (II.5)$$

Je höher die Verzögerung ("Delay") und je schmalbandiger die Leitung, desto höher ist das Gewicht der Kante E. Je höher das Gewicht der Kante, desto unattraktiver ist sie für die Verwendung im Verteilungsbaum. Die Parameter a und b müssen entsprechend der gewünschten Eigenschaften des MSBs festgelegt werden. Soll beispielsweise der Baum auf minimale Verzögerung, daß heißt möglichst schnelle Zustellung der Events, optimiert werden, so muß $a=1$ und $b=0$ gesetzt werden. Soll der Baum möglichst geringe Kosten aufweisen, so muß $a=0$ und $b=1$ gesetzt werden.

Leider ist die Bestimmung der maximalen Kapazität einer Leitung ohne genaue Kenntnis der Netztopologie nicht möglich, da aus der Anwendungsschicht nicht feststellbar ist, wieviel anderer Netzwerkverkehr die betrachtete Leitung belastet. Für eine Implementierung folgt daraus, daß nur die Kosten und eventuell die Leitungsqualität (→Paketverluste) in die Berechnung eingehen können.

Mitgliederfluktuation

Dieses Maß beschreibt die Fluktuation von Mitgliedern der Gruppe pro in der Gruppe gesendetem Event. Damit kann bestimmt werden, wie oft die Kommunikationsstruktur des Eventmanagers geändert werden muß. Je höher die Fluktuation, desto mehr lohnen sich Ansätze, die den spannenden Baum nicht in seiner Gesamtheit neu berechnen, sondern inkrementell neue Knoten einbinden und eventuell alte, nicht mehr benötigte Knoten aus dem Baum entfernen.

Die Mitgliederfluktuation läßt sich in drei Kategorien unterteilen:

- **Hinzukommen**
  ein Knoten tritt der Gruppe das erste Mal bei.

$$Flukt_{dazu} = \frac{\#Plus\text{-}Operationen}{\#Events_{gesendet}} \tag{II.6}$$

- **Verlassen**
  ein Knoten verläßt die Gruppe.

$$Flukt_{weg} = \frac{\#Minus\text{-}Operationen}{\#Events_{gesendet}} \tag{II.7}$$

- **Umziehen**
  ein Knoten verläßt die Gruppe und dafür steigt ein neuer Knoten in die Gruppe ein. Dies entspricht der Migration eines Consumers vom ersten Knoten auf den zweiten Knoten unter der Voraussetzung, daß der Consumer auf beiden Knoten das einzige Mitglied dieser Gruppe ist, denn nur dann muß die Kommunikationsstruktur des Eventmanagers geändert werden.

$$Flukt_{Migrat} = \frac{\#Migrationen}{\#Events_{gesendet}} \tag{II.8}$$

Die Mitgliederfluktuation hat einen hohen Einfluß auf die Effizienz von Änderungen der Kommunikationsstruktur des Eventmanagers. Insbesondere das Umziehen von einem Knoten zu einem anderen tritt in einem System mobiler Agenten sehr häufig auf und muß daher besonders beachtet werden.

In Abbildung II.1 ist dieser Vorgang graphisch dargestellt.

Gruppenaktivität

Dieses Maß beschreibt, wieviele Events pro Zeiteinheit in der Gruppe versendet werden und ist ein Indikator für die Aktivität der Gruppe. Je mehr Events versendet werden, desto aktiver ist die Gruppe und desto intensiver ist die Kommunikation zwischen den einzelnen Teilnehmern. Je höher die Gruppenaktivität ist, desto häufiger lassen sich "normale" Events mit Steuerinformationen eines Gruppenkontrollprotokolls kombinieren ("Piggybacking").

$$Aktivität_{Gruppe} = \frac{\#Events_{gesendet,\ Gruppe}}{\Delta t} \qquad (II.9)$$

Δt bezeichnet das betrachtete Zeitintervall. Die folgende Formel *II.10* beschreibt die Aktivität eines einzelnen Suppliers.

$$Aktivität_{Supplier} = \frac{\#Events_{gesendet}}{\#Events_{gesendet,\ Gruppe}} \qquad (II.10)$$

$Aktivität_{Supplier}$ ist ein Maß dafür, wie stark ein bestimmter Supplier zum Eventverkehr innerhalb der Gruppe beiträgt. Je ausgeglichener dieser Wert für alle Supplier einer Gruppe ist, desto billiger können unter Umständen Gruppenkontrollprotokolle implementiert werden, weil Kontrollinformationen mit einer regulären Nachricht mitgeschickt werden können. Der Wertebereich liegt zwischen 0 (der betrachtete Supplier sendet überhaupt nichts) und 1 (der betrachtete Supplier versendet alle Events der Gruppe).

Umfassende Kostenuntersuchungen von Protokollen können erst mit Kenntnis der verwendeten Algorithmen angestellt werden.

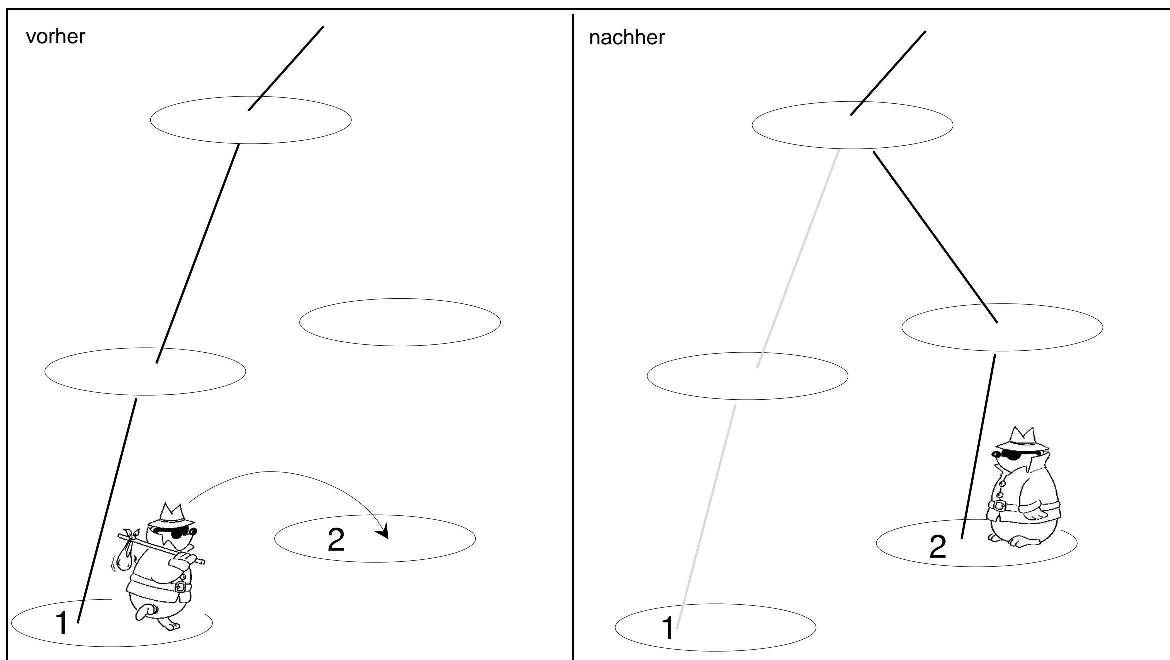


Abb. II.1: Erläuterung der Formel (II.8). Agent A migriert von Knoten 1 nach 2. Die Kommunikationspfade des Eventchannels sind angegeben. Unter den Annahmen, daß die Kommunikationsstruktur möglichst billig bleiben soll und der Agent der einzige Consumer des Eventchannels auf den beteiligten Knoten ist, muß der Eventmanager nun die Verbindung zu Knoten 1 entfernen und dafür eine Verbindung zu Knoten 2 aufbauen.

II.3. Terminierung

Der Begriff Terminierung wird in vielen Zusammenhängen gebraucht. Im Rahmen dieser Arbeit beschränken wir uns auf die Anforderungen der expliziten Terminierung von mobilen und stationären Agenten. Die in der Literatur (zum Beispiel in [Raynal88]) häufig anzutreffenden Algorithmen zur Feststellung, ob ein verteilter Algorithmus terminiert ist, verwenden irgendeine Form von Eltern-Kind-Beziehung zwischen den beteiligten Prozessen. In einem Agentensystem können aber auch Agenten, deren Erzeugeragent nicht mehr existiert, noch eine sinnvolle Aufgabe erfüllen, da sie autonom und kooperativ im System arbeiten können. Der Begriff "Terminierung" bezeichnet in dieser Arbeit den Vorgang des ausdrücklichen Beendens der Aufgabe und aller Agenten, die an ihrer Lösung arbeiten.

Im nachfolgenden Kapitel II.4 werden Verfahren zur Waisenerkennung vorgestellt, mit denen die Lebensberechtigung eines Agenten geprüft werden kann, und ob eine Terminierung eingeleitet werden muß.

II.3.1. Explizite Terminierung von Agenten

Zur Ermittlung von Anforderungen wird ein Anwendungsfall konstruiert und untersucht, welche Bedingungen der Eventmanager dafür erfüllen muß.

Ein Beispiel

Wie im ersten Teil dieser Arbeit (Kapitel I.4) dargelegt, werden zur Erfüllung einer Aufgabe oft viele Agenten auf einmal gestartet und in die Welt geschickt. Dieses Szenario ist zum Beispiel für Informationsrecherche naheliegend. Angenommen es genügt, wenn einer dieser Agenten zum Heimatort zurückkehrt und seine Ergebnisse meldet. Dann verursachen alle übrigen Agenten, die noch durch die Welt migrieren, unnötig Kosten für Rechenzeit, Kommunikation und Migration. In Anbetracht einer potentiellen kommerziellen Nutzung und aus Sicht der Betreiber der verbrauchten Ressourcen ist das nicht erwünscht, da der Benutzer bestimmt nicht für Kosten, die **nach** Erhalt des Ergebnisses auflaufen, bezahlen will.

Ein Protokoll zur Lösung des Problems

Alle Agenten der Aufgabe sind Consumer im selben Eventchannel. Nachdem der erste Agent sein Ergebnis am Heimatort abgeliefert hat, wird von der Anwendung ein Terminierungsevent in den Eventchannel gesendet. Alle Agenten terminieren sich, sowie der Terminierungsevent eintrifft.

Was gibt es denn da für besondere Anforderungen?

Das Protokoll ist zwar sehr einfach, stellt aber grundlegende Anforderungen an den Eventmanager.

- **Zuverlässigkeit**
  Der Eventmanager muß den Terminierungsevent an jeden betroffenen Agenten ausliefern. Daraus folgt, daß der Eventmanager die Auslieferung dieser Events sehr aufwendig kontrollieren und absichern muß, damit sie auch bei fehlerhafter Übertragung, möglichen Leitungsausfällen oder gar Ausfällen von Knoten noch ihr Ziel erreichen (siehe dazu auch Kapitel II.8).

- **Mobilität der Consumer**
  Wenn die Behandlung der Terminierungsevents wie beschrieben gelöst wird, fällt auf, daß ein Consumer möglicherweise gerade zwischen zwei Locations migriert, wenn der Event eintrifft (so er denn überhaupt an einer der beiden Locations eintrifft, denn genau genommen ist der Agent weder an der einen, noch an der anderen Location, sondern dazwischen ...). Die Problematik ist etwas umfangreicher und daher ausführlich in Kapitel II.6 besprochen.
  Man könnte die Behandlung der Terminierungsevents auch den Locations überlassen und hätte dann das Problem der mobilen Consumer nicht. Dann muß aber jeder Terminierungsevent an jede Location des Agentensystems gesendet werden, da sich der Agent potentiell überall befinden kann. Dies entspricht im Prinzip einem weltweiten Broadcast und ist aufgrund der benötigten Bandbreite und der großen Menge an Terminierungsinformationen, die eine Location zu speichern hat, nicht akzeptabel.

Wir benötigen zur Realisierung dieses Protokolls einen Reliable Multicast, der mit mobilen Consumern umgehen kann.

Verfeinerung

Das Ziel der Terminierungsevents ist, möglichst schnell eine Gruppe von Agenten verschwinden zu lassen. Angenommen der Eventchannel der Gruppe wird auch für andere Funktionen der Anwendung (zum Beispiel Synchronisierung) verwendet. Dann kann es vorteilhaft sein, die Terminierungsevents bevorzugt auszuliefern und so eine schnellere Verarbeitung zu erreichen. Mit folgenden Verfahren läßt sich das erreichen.

- **Prioritäten**
  Events werden Prioritäten zugeordnet. Es werden immer zuerst Events der höchsten Priorität ausgeliefert, die gerade verfügbar ist. Terminierungsevents werden dann einfach mit einer sehr hohen Priorität versendet und dadurch bevorzugt ausgeliefert.

- **Expedited Data**
  Eine Vereinfachung der Prioritäten. Es gibt nur zwei Prioritätsstufen: "Normal" und "Bevorzugt". Bevorzugte Events werden bei Eintreffen sofort bearbeitet, während normale Events die regulären Bearbeitungswege (zum Beispiel für Reihenfolgesicherung) durchlaufen.

Für Terminierung ist Expedited Data vollkommen ausreichend und auch sehr einfach zu implementieren.

Kosten

Durch das Protokoll entstehen, außer den Verwaltungskosten des Eventchannels bei Migrationen der Agenten, lediglich die Kommunikationskosten für das Versenden **eines** Events.

Nur für den Zweck der Terminierung ist ein Eventmanager nicht nötig. Stattdessen ist es einfacher, ein System verteilter Objektreferenzen zu verwenden (zum Beispiel SSP-Chains [Shapiro92]) und unter Verwendung dieser Referenzen die Terminierungsmethode des Agenten entfernt aufzurufen.

II.3.2. Implizite Terminierung mit Agentengruppen

Wir wollen das Beispiel aus dem vorigen Abschnitt nun mit einer Terminierungsgruppe modellieren. Die Aufgabe entspricht einer ODER-Gruppe. Irgendeiner der Agenten muß seine Aufgabe erledigt haben, damit der Gruppenkoordinator (internal GC) die ODER-Gruppenbedingung "Ein Agent muß fertig sein" zu wahr evaluiert und die Gruppe terminiert.

Auf den ersten Blick entspricht dies der expliziten Terminierung. Konzeptionell besteht aber ein Unterschied zwischen den Verfahren. Bei der expliziten Terminierung ist ein Agent so lange aktiv, bis er einen Terminierungsevent erhält. Es ist irrelevant, von wem oder wie er den Event erhält. Bei der impliziten Terminierung definiert sich die Lebensberechtigung des Agenten aus dem Vorhandensein seiner Terminierungsgruppe und dem nicht gesendeten Terminierungsevent. Erhält der Agent den Terminierungsevent oder ist seine Gruppe nicht mehr vorhanden, so muß er terminiert werden. Die Überwachung der Terminierungsbedingung wird vom GC des Agenten durchgeführt. Die implizite Terminierung behandelt automatisch auch Waisen und ist somit eine Kombination aus reiner Terminierung und reiner Waisenerkennung und -behandlung. In Kapitel II.4 werden die Hintergründe von Waisenerkennung und -behandlung ausführlicher vorgestellt.

Woran erkennt ein verteilt implementierter GC, ob eine Gruppe noch vorhanden ist? Grundmerkmal einer Gruppe ist neben dem GC auch der gruppeneigene Kommunikationskanal, der vom Eventmanager verwaltet wird. Wir können sicherstellen, daß der Kanal vom erzeugenden Agenten eine eindeutige Bezeichnung erhält (zum Beispiel: Agentenname + Locationname + lokale Uhrzeit). Unter Angabe dieses Namens kann ein GC jederzeit beim Eventmanager nachfragen, ob der Kanal noch existiert. Ist dies nicht der Fall, so ist entweder ein schwerer Fehler aufgetreten, der zum Zusammenbruch des Kanals geführt hat, oder die Gruppe wurde terminiert und der zugehörige Kanal geschlossen. In beiden Fällen bleibt dem GC nichts anderes übrig, als seinen Agenten zu terminieren. Entweder war der Fehler so schwerwiegend, daß ein Weiterarbeiten nicht sinnvoll ist oder die Gruppe bereits terminiert und der Agent ist in Wirklichkeit ein Waise und hat den Terminierungsevent nicht erhalten.

Anforderungen an den Eventmanager

Die Anforderungen an den Eventmanager sind ähnlich gelagert wie bei der expliziten Terminierung. Mit einem Unterschied: Es ist nun nicht mehr zwingend notwendig, daß ein Terminierungsevent einen Agenten unbedingt erreichen muß. Es genügt der "best-effort"-Ansatz. Der Eventmanager versucht den Event zuzustellen. Wenn dies aufgrund einer Netzwerkpartitionierung nicht möglich ist, oder sich ein Gruppenteilnehmer auf einer Location in einem mobilen Rechner befindet, kann der GC des Agenten bei Bedarf feststellen, ob die Gruppe noch existiert und entsprechend handeln.

II.3.3. Folgerungen

Die explizite Terminierung ist verhältnismäßig einfach, vorausgesetzt man hat einen zuverlässigen Eventmanager, der die Terminierungsevents an jeden Agenten auch wirklich ausliefert. Das Nachführen der Kommunikationsstrukturen des Eventmanagers bei Migrationen des Agenten ist allerdings im Verhältnis zur Nutzung dieser Strukturen sehr teuer.

Die implizite Terminierung bietet unter Verwendung der Agentengruppen ein einfaches und zugleich mächtiges Verfahren zur Beschreibung des Terminierungsverhaltens von Agenten. Das vorgestellte Beispiel für eine ODER-Gruppe läßt sich ohne großen Aufwand auf andere Gruppenarten anpassen, da lediglich die Terminierungsbedingung im internal GC geändert werden muß. Unter Verwendung der impliziten Terminierung läßt sich auch die Waisenerkennung behandeln. Im folgenden Kapitel gehen wir näher darauf ein.

II.4. Waisenerkennung

In diesem Kapitel wird die Erkennung und Behandlung von Waisen (engl. Orphans) untersucht. Die vorgestellten Protokolle sind jeweils als anschauliches Arbeitsbeispiel gedacht. Zu jedem vorgestelltem Algorithmus stellen wir die daraus resultierenden Anforderungen vor und geben Abschätzungen zu Kosten und Skalierbarkeit an.

II.4.1. Waisen

Wann ist ein Agent eine Waise? Zepf schreibt in seiner Studienarbeit [Zepf96], daß ein Agent eine Waise ist, wenn der Agent nicht mehr lebensberechtigt, das heißt entweder vergessen oder überflüssig ist. Wenn ein Agent lebensberechtigt ist, erfüllt er in irgendeiner Hinsicht eine sinnvolle Aufgabe. Irgendeine Kontrolleinheit dieser Aufgabe hat den Agenten nicht vergessen und somit ist er nicht überflüssig.

Was soll mit dem Agenten passieren, wenn er doch vergessen wurde? Oder die Aufgabe nicht mehr existiert, da ihre Berechnung beendet ist und er dies aufgrund von Netzwerkproblemen nicht mitbekommen hat? Ziel der Waisenerkennung (engl. "Orphan Detection") ist es, solche Agenten zu finden und normalerweise möglichst schnell zu terminieren.

Wer bestimmt Waisen?

Wie bei der Terminierung auch, basieren alle klassischen Algorithmen zur Waisenerkennung auf den Eltern-Kind-Beziehungen zwischen Prozessen. Wenn der Elternprozeß beendet wird, haben auch die Kindprozesse, die beispielsweise durch einen RPC auf einem Server angestoßen wurden, keine Lebensberechtigung mehr (siehe dazu beispielsweise [Panzieri88]). In einem Agentensystem ist diese einfache Sichtweise nicht mehr gültig, da Kindagenten durchaus noch eine sinnvolle Aufgabe erfüllen können, auch wenn die Eltern bereits seit langem beendet sind. Die Waisenerkennung darf daher nicht auf Basis der Eltern-Kind-Beziehungen agieren! Jeder Agent erfüllt eine Aufgabe. Erst wenn diese Aufgabe nicht mehr sinnvoll erfüllt werden kann oder bereits von einem oder mehreren anderen Agenten erfüllt wurde, ist der Agent eine Waise.

Meist sind Agenten zur Erfüllung ihrer Aufgabe auf Dienste des Agentensystems angewiesen. Wenn ein Dienst nicht mehr verfügbar ist, so kann der Agent seine Aufgabe nicht mehr erfüllen. Auch dann ist er genaugenommen eine Waise.

Die oberste Ebene

Nun ist der Begriff "Aufgabe, die von einem Agenten ausgeführt wird" nicht besonders aussagekräftig. Da Waisenerkennung immer mit Referenzen arbeitet, wird eine Referenz auf die "Aufgabe" des Agenten benötigt. Wenn die Aufgabe existiert, darf der Agent weiterleben, ansonsten wird er terminiert. Zepf verwendet als Platzhalter für die Aufgabe sogenannte Schatten, die eine außerhalb des Agentensystems laufende Anwendung und

deren Aufträge an das Agentensystem repräsentieren [Zepf96]. Solange die extern laufende Anwendung sich "merkt", welche Schatten sie wo erzeugt hat, ist das kein Problem. Wenn sie durch einen Absturz oder einen Programmfehler die angelegten Schatten nicht mehr löscht, dürfen die zugehörigen Agenten für immer im System bleiben, obwohl sie in Realität Waisen sind. Die Waisenerkennung des Agentensystems, ausgelegt für viele Fehler und Kommunikationsprobleme, kann diesen simplen Fehler nicht behandeln. Es ist Sache der Anwendung solche Schatten zu finden und zu vernichten.

Eine andere Sichtweise hat sich aus der Praxis des Agentensystems MOLE ergeben. Die Anwendung wird ebenfalls durch einen Agenten realisiert, der mit dem Benutzer zum Beispiel über Java-Applets kommuniziert. Diese Agenten **sind** die Anwendung und dürfen nicht von der Waisenerkennung erfaßt werden. Wie auch bei den Schatten, können Programmfehler in diesen Agenten (wohlgemerkt in den *Agenten*, nicht im System!) dafür sorgen, daß die ganze Waisenerkennung als Systemdienst nicht mehr funktioniert.

Das Problem von Waisenerkennung auf der obersten Ebene ist wahrscheinlich nicht automatisch lösbar, da jede Aufgabe eines Agenten irgendwann einmal von einem menschlichen Anwender des Systems gestellt wurde. Nur der Anwender kann bestimmen, ob er an den Ergebnissen einer Berechnung noch interessiert ist oder nicht.

Das Problem ist lösbar, wenn man den Anwender in die Waisenerkennung miteinbezieht. Beispielsweise per Telepathie oder indem man alle 10 Sekunden nachfragt, ob er noch an dem Ergebnis interessiert ist. Diese Verfahren sind allerdings zur Zeit für Computer noch nicht möglich beziehungsweise für den Anwender nicht besonders komfortabel und werden daher von uns nicht weiter verfolgt.

Wann können Waisen entstehen? Ein Beispiel

Ein Agent, nennen wir ihn Makler, sitzt an einem Ort und erhält unregelmäßig von zwei entfernten Diensten per Nachrichtenkommunikation unterschiedliche Informationen über ein gemeinsames Thema, zum Beispiel Börsenkurse und Bilanzdaten eines Unternehmens

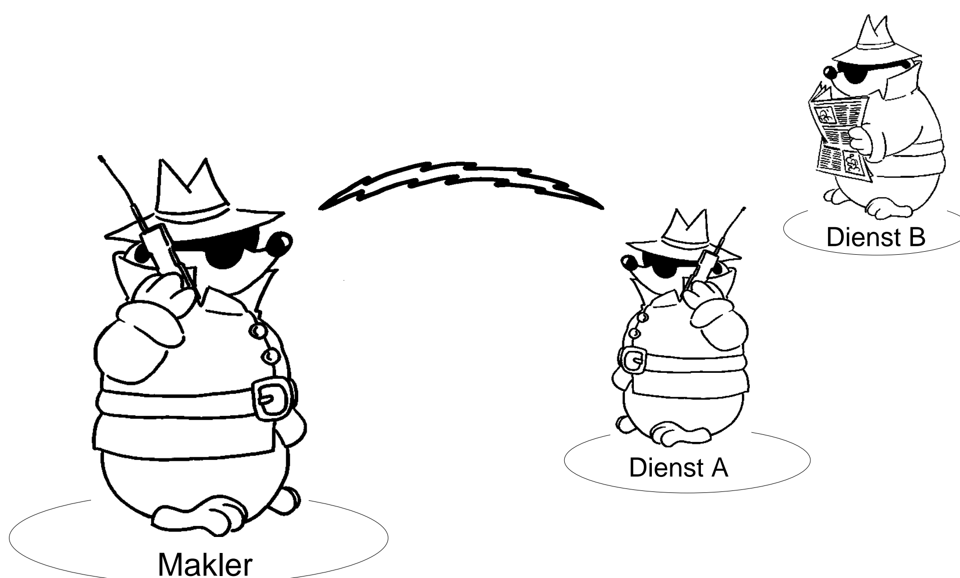


Abb. II.2: Der Makler bezieht unregelmäßig Informationen von Dienst A und B. Wann ist der Makler eine Waise?

(Abbildung II.2). Auf Basis dieser Informationen kauft und verkauft er Aktien (sozusagen ein verbotener Insiderhandel). Sollte einer dieser beiden Dienste nicht mehr verfügbar sein, hat der Makler keine gesicherten Informationen mehr und kann seine Aufgabe nicht mehr erfüllen. Dann ist er eine Waise, obwohl seine Aufgabe weiterhin besteht, er also sinnvoll arbeiten kann, sowie er neue Informationen erhält. Da die Informationen unregelmäßig eintreffen, bedeutet eine lange Pause zwischen zwei Sendungen eines Dienstes nicht, daß der Agent jetzt arbeitslos ist.

Das Beispiel zeigt, daß sich Waisenerkennung in einem Agentensystem eher in einem chaotischen Beziehungsgeflecht, als in einem sauber hierarchisch strukturierten Baum abspielt. Ziel der Waisenerkennung ist das Überprüfen dieses Beziehungsgeflechts auf Ausfälle.

Natürlich ist diese Sicht der Dinge nicht ganz unproblematisch. Ein Dienst kann zwar (beispielsweise durch einen Rechnerzusammenbruch) ausfallen, aber nach verhältnismäßig kurzer Zeit auch wieder verfügbar sein (nämlich dann, sowie der Rechner neu gestartet wurde). Wenn in der Zwischenzeit die Waisenerkennung aktiv wurde, ist der Agent terminiert, obwohl er inzwischen weiterleben dürfte. Im Idealfall sollten Verfahren zur Waisenerkennung dies berücksichtigen.

Für die folgenden Betrachtungen wird vorausgesetzt, daß ein Agent seine Beziehungen zu Eltern und Kommunikationspartnern kennt. Wenn im folgenden von Beziehungspartnern die Rede ist, so sind damit sowohl Eltern als auch Kommunikationspartner gemeint.

Alle Verfahren zur Waisenerkennung verwenden Timeouts, nach deren Ablauf bei einer übergeordneten Instanz nachgefragt wird, ob der Agent weiterleben darf. So ähnlich sich die Verfahren sind, so stark unterscheiden sie sich in Details.

II.4.2. Aktive Waisenerkennung

Bei dieser Herangehensweise stellt der Agent selber die Frage "Bin ich eine Waise?". Ziel ist es, möglichst einfach und billig herauszubekommen, ob der Agent noch eine Lebensberechtigung hat.

Ein mögliches Protokoll

Kehren wir zum Beispiel zurück. Der Agent Makler schickt, wie alle anderen Teilnehmer des Channels auch, nach Ablauf eines Timeouts *Timeout* in den mit seiner Aufgabe assoziierten Channel den Event "Makler is alive". Empfängt einer seiner Beziehungspartner, die Dienste A und B, diesen Event, so setzt dieser seinen Prüf-Timeout für Makler *TimeoutCheck(Makler)* neu auf. Andere in den Channel gesendete Events eines Teilnehmers *a* gelten ebenfalls als Lebenszeichen dieses Teilnehmers und führen zum Neuaufsetzen des entsprechenden Timeouts *TimeoutCheck(a)* . Vom Agenten oder Dienst selbst gesendete Events führen zum Neuaufsetzen des eigenen Timeouts *Timeout*.

```
// Code wird von Objekt a ausgeführt. Es können weitere
// Threads existieren, die Events in den Channel verschicken
// Nebenbedingung: selfWait < checkWait
for any partner j do
    set TimeoutCheck(j) = checkWait;
next

repeat forever
    if Timeout is expired
        post "a is alive" to channel;
        set Timeout = selfWait;
    end if
    for any partner j do
        if TimeoutCheck(j) is expired
            // Partner hat sich innerhalb des Timeouts nicht
            // gemeldet. Fehlerbehandlung starten.
        end if
    next
loop


// Code wird von Objekt a beim Senden eines Events
// ausgeführt
set Timeout = selfWait;


// Code wird von Objekt a bei Empfang
// eines Events ausgeführt
case event type of
    "b is alive":
        if b is partner
            set TimeoutCheck(b) = checkWait;
        end if

    any event from b:
        if b is partner
            set TimeoutCheck(b) = checkWait;
        end if
end case
```

Abb. II.3: Pseudocode eines einfachen Protokolls zur Waisenerkennung unter Verwendung von Events. Jeder Teilnehmer eines Channels führt diesen Algorithmus aus.

Mit anderen Worten: Jedes Gruppenmitglied gibt in regelmäßigen Abständen ein Lebenszeichen von sich, das von den anderen Gruppenmitgliedern aufgefangen wird. Das Protokoll ist als Pseudocode mit allen im folgenden besprochenen Erweiterungen in Abbildung II.3 gegeben.

Da jeder Agent einer Aufgabe die "x is alive"-Events in den zugeordneten Channel schickt, müssen auch sämtliche Beziehungspartner Teilnehmer dieses Channels sein. Wenn zwei unterschiedliche Agenten dieselben Beziehungspartner haben, erhalten alle Agenten einer Aufgabe zumindest von Elternseite regelmäßig die Bestätigung weiterzuleben. Da das Sendeverhalten der Beziehungspartner nicht voneinander abhängt, muß jedes Objekt für jeden Beziehungspartner einen eigenen Timeout unterhalten.

Wenn der Agent bei den eigenen "x is alive"-Events seinen aktuellen Aufenthaltsort noch gleich mitschickt (zum Beispiel IP-Adresse:Port seiner Engine), ist nebenbei ohne zusätzliche Kosten ein im Rahmen des Intervalls Timeout *genaues System zum Finden von Agenten vorhanden. Mit einigen zusätzlichen Maßnahmen tritt dieser Effekt bei allen Mechanismen zur Waisenerkennung auf.*

Sicherheit

Ohne weitere Maßnahmen kann ein Agent beim Empfang von Events nicht feststellen, von wem der Event stammt. Wenn ein Spion mindestens einen Beziehungspartner des Agenten Makler kennt, so könnte er zum Beispiel anstelle von Dienst A antworten. Wenn ein Agent nur von seinen Eltern abhängt, so könnte ein anderer Abkömmling dieser Eltern die Antwort anstelle der Eltern senden. Angenommen, zwei Agenten arbeiten zusammen, so können sie sich gegenseitig die Bestätigungen zum Weiterleben über den Eventchannel senden. Allerdings erhält jeder Beziehungspartner diese Events, auch die eigentlich zuständigen Eltern, die dann geeignete Maßnahmen ergreifen können. Zum Beispiel indem sie eine explizite Terminierung für diese zwei Doppelagenten verschicken.

Wie bei der Terminierung kann man die Funktionalität der Waisenerkennung in den Agentenbasisklassen realisieren. Standardmäßig wird die Beziehung zu einem Repräsentanten der Aufgabe des Agenten geprüft. Die Liste der Beziehungspartner kann vom Agentenprogrammierer dann entsprechend um die gewünschten Beziehungspartner erweitert werden. Der Programmierer kann die Waisenerkennung aber nicht verhindern oder unterlaufen.

Bestimmung von Timeout und TimeoutCheck

Die beiden Timeouts *Timout* und *TimeoutCheck* müssen für alle Agenten der Gruppe gleich sein. Welchen Wert sie aber haben sollen, kann nicht allgemein festgelegt werden. Nur der Programmierer weiß, wie schnell seine Agenten durch die Waisenerkennung erfaßt werden sollen. Dabei muß berücksichtigt werden, daß die Verzögerung der Events im Netz und die Verarbeitungszeit in den Knoten schwanken kann.

Natürlich löst dieser Ansatz nicht das eigentliche Problem, sondern reicht den Schwarzen Peter nur an den Agentenprogrammierer weiter, der sinnvolle Werte für die Timeouts festlegen muß. Dies ist aber trotzdem akzeptabel, da ein Timeout im Grunde genommen nichts anderes als ein Hilfsmittel darstellt, um unbeantwortbare Fragen in einem vertretbaren Zeitrahmen (daß heißt nach Ablauf des Timeouts) nicht unbedingt korrekt, aber zumindest eindeutig zu entscheiden.

Einfluß von Migration

Bisher ist bei der Beschreibung des Protokolls die Agentenmigration noch gar nicht berücksichtigt. Angenommen der Makleragent migriert zu einer anderen Location. Während der Migration wird ein Event eines seiner Kommunikationspartner versendet. Je nachdem wie der Eventmanager Events an migrierende Agenten behandelt, erreicht dieser Event den Makler unter Umständen nicht und ist verloren. Eine einfache (aber nicht besonders sinnvolle) Lösung wäre, einem Agenten, der an Eventkommunikation teilnimmt,

die Migration zu verbieten. Dies widerspricht natürlich dem Wesen eines Systems mobiler Agenten und ist daher nicht akzeptabel. In Kapitel II.6 wird dieses Thema näher beleuchtet.

Anforderungen an den Eventmanager

Zusammenfassend die Anforderungen an den Eventmanager bei Verwendung von aktiver Waisenerkennung nach dem vorgestellten Modell:

- **einigermaßen zuverlässig**
  Es ist nicht notwendig, daß jeder "x is alive"-Event bei jedem Partner eintreffen muß. Bei entsprechend lockerer Handhabung der Timeouts und mehrfachem Senden der entsprechenden Events, ist das akzeptabel, da jeder Teilnehmer der Gruppe die Requests und Bestätigungen sowohl selbstständig, regelmäßig, als auch auf Anforderung versendet. Das heißt es können Events verloren gehen, ohne daß die Funktionalität leidet! Wieviel Events in Reihe verloren gehen dürfen, hängt zum einen vom Timeout für die Terminierung *Timeout(a)* und zum anderen vom Timeout des Nachfragens *TimeoutCheck(j)* ab. Wenn *Timeout(a)* doppelt so groß ist wie *TimeoutCheck(j)*, kann genau eine Bestätigung zwischen zwei erfolgreichen Kommunikationen verloren gehen und das System funktioniert immer noch.

- **Absenderkennungen**
  Um den Mißbrauch der Spione aus dem obigen Beispiel zu verhindern, braucht ein Event eine eindeutige Absenderkennung, die nicht vom Agenten geändert werden kann. Prinzipiell kann jedes Objekt des Agentensystems den Eventmanager verwenden und Events versenden. Bestimmt man die Absenderkennung zum Beispiel nur auf Basis des Agentennamens, so schließt man alle Server und Grundkomponenten des MOLE-Kerns vom Eventmanager aus. Dies ist allerdings nicht tragisch, da diese Komponenten ohne weiteres einen stationären Systemagenten instanziieren können, unter dessen Namen dann die Eventkommunikation durchgeführt wird (ähnlich zum Moleview-Agenten von Moleview [Beck96]). Die Gründe werden in Kapitel II.6.3 besprochen.

Und was kostet das?

Für jeden Teilnehmer eines Channels wird im Intervall *Timeout* mindestens eine Nachricht gesendet. Das ist entweder ein normaler Event, den der Agent abschickt oder der "x is alive"-Event.

An Kosten für das Versenden eines Events in einem spannenden Baum fällt die Summe der Kosten der Gewichte der Kanten des Baums an. Die Verarbeitungszeit der Nachrichten in den einzelnen Knoten sei hier vernachlässigt (der Baum besteht bei n Knoten aus n-1 Kanten):

$$Kosten_{Event} = \sum_{i=1}^{m} W_i \qquad (II.11)$$

mit

$m = n$-1, Anzahl der Kanten im Baum,
$W_i = w(E_i)$, Kostenfunktion über Kante i.

Die höchsten zusätzlichen Kosten $Kosten_{aOD}$ entstehen, wenn innerhalb des Timeoutintervalls von keinem Teilnehmer des Channels ein Event erzeugt wird ("worst case"). Dann muß jeder Teilnehmer einen "x is alive"-Event erzeugen. Im besten Fall sendet jeder Teilnehmer des Channels innerhalb des Timeouts mindestens einen regulären Event. Der durchschnittliche Fall hängt von der Aktivität der Eventsupplier in der Gruppe ab.

Die zusätzlichen Kosten durch die Waisenerkennung in einem Intervall *Timeout* betragen:

$$\begin{aligned} Kosten_{aOD,\ WorstCase} &= \#Teilnehmer \times Kosten_{Event} \\ Kosten_{aOD,\ AvgCase} &= \#Teilnehmer \times Kosten_{Event} \times Aktivität_{Gruppe} \\ Kosten_{aOD,\ BestCase} &= 0 \end{aligned} \qquad (II.12)$$

Die Skalierbarkeit des Verfahrens hängt sehr stark von der Aktivität der Gruppe ab. Je mehr die Teilnehmer des Channels über Events miteinander kommunizieren, desto weniger Events müssen zusätzlich für die Waisenerkennung erzeugt werden. Die zusätzlichen Kosten durch das Protokoll steigen bei zunehmender Gruppengröße linear an.

II.4.3. Passive Waisenerkennung

In diesem Abschnitt wird Waisenerkennung von der "anderen" Seite her betrachtet. Im Gegensatz zur aktiven Waisenerkennung, muß der Agent nun nicht selber feststellen, ob er eine Waise ist. Stattdessen existiert bei der passiven Waisenerkennung eine Kontrollinstanz (zum Beispiel in der Location), die die Agenten überwacht und regelmäßig (oder aufgrund von Ereignissen) prüft, ob ein Agent zu einer Waisen geworden ist.

Ein Protokoll

Jede Location überwacht die Lebensberechtigungen der Agenten. Zu diesem Zweck hat jeder Agent nur eine begrenzte Lebensdauer. Die Location muß sich um eine Bestätigung des Auftraggebers bemühen. Zu diesem Zweck existiert pro Aufgabe ein eigener Eventchannel. Die Teilnehmer dieses Eventchannels sind alle Locations, auf der Agenten der Aufgabe laufen, und alle Objekte, die von Agenten für diese Aufgabe erzeugt werden. Die Location sendet, kurz bevor der Agent das Ende seiner Lebenszeit erreicht, einen Request-Event mit dem Namen des Agenten als Parameter. Eine Kontrollinstanz der Aufgabe (das müssen nicht notwendigerweise die Eltern sein), antwortet mit einem Grant-Event, woraufhin die Location die Lebenszeit des Agenten verlängert. Jede Kontrollinstanz der Aufgabe muß dafür eine Liste aller jemals in ihrem Namen erzeugten Agenten führen. Falls der Auftraggeber ein mobiler Agent ist, muß er diese Liste natürlich bei jeder Migration mitnehmen.

Da sehr wahrscheinlich alle Agenten der Gruppe die gleiche Lebenszeit haben, wird der Eventchannel jeweils kurz vor dem Ende des Lebenszeitintervalls mit Request- und Grant-Events überflutet. Um dies etwas zu entzerren, wird der Zeitpunkt, zu dem die Location den Request-Event abschickt, zufällig innerhalb eines kleinen Zeitraums vor dem Ende der Lebenszeit bestimmt.

Der geprüfte Agent wird terminiert, wenn seine Lebenszeit abgelaufen ist.

Sicherheit

Wie auch schon bei dem Protokoll aus Abschnitt II.4.2 kann ein beliebiger Teilnehmer des Channels den Grant-Event verschicken. Um dem entgegenzuwirken, muß der Agent so erweitert werden, daß die Location in ein geschütztes Feld den Verantwortlichen für den Agenten einträgt. Bei Empfang des Grant-Events kann sie die Absenderkennung mit der Erzeugerkennung vergleichen und somit feststellen, ob der Grant-Event vom richtigen Objekt kommt.

Verbesserungen

Der Erzeuger des Agenten kann mobil sein, die aus der Mobilität entstehenden Probleme werden in Kapitel II.6 behandelt. Im folgenden werden einige Verbesserungen des Basisprotokolls besprochen.

- **Behandlung des Grant-Events und Netzwerkbelastung**
  Die Grant-Events können prinzipiell auf zwei Arten zu der anfragenden Location gesendet werden. Die eine Möglichkeit ist mit direkter Kommunikation zwischen Auftraggeber und dem Agenten. Sie hat den Vorteil, daß das Netzwerk nur auf der Route zwischen Sender und Empfänger mit dieser Nachricht belastet wird. Der Auftraggeber kann so auch sehr fein steuern, welche Agenten weiterleben dürfen. Nachteilig ist allerdings, daß dann der Agent bis zum Ende der Prüfung nicht migrieren darf. Die andere Möglichkeit ist, den Grant-Event wieder in den Eventchannel zu schicken. Das hat den Vorteil, daß nicht nur der Agent, dessen Lebenszeit zuende geht mitbekommt, daß er weiterleben kann, sondern auch alle anderen Agenten, die von diesem Auftraggeber erzeugt wurden. Auch alle anderen Locations, die Agenten dieser Aufgabe haben und damit Consumer am selben Eventchannel sind, bekommen die Anfrage mit und brauchen nicht selber Request-Events zu senden. Dieser Ansatz kann gut für Agentengruppen verwendet werden.

- **Verkleinern von migrierenden Auftraggebern**
  Der Auftraggeber nimmt, wenn er an einer Location viele Agenten erzeugt hat, nicht die ganze Liste seiner Kinder mit. Er erzeugt stattdessen einen Proxy-Agenten, der die Liste verwaltet und nimmt nur den Verweis auf diesen Proxy-Agenten mit. Das System läßt sich auch hierarchisch einsetzen, indem beim nächsten Proxy-Agenten der Verweis auf den ersten Proxy-Agenten bleibt. Der Auftraggeber muß bei einer Migration deshalb nur den Verweis auf den letzten Proxy-Agenten mitnehmen. Für Anwendungen, die das System mit Agenten vollpumpen ("Flooder") ist diese Verbesserung sicher

brauchbar. Natürlich muß die Absenderkennung des Proxy-Agenten mit der des Auftraggebers identisch sein, damit der Grant-Event beim Empfänger auch akzeptiert wird. Eine andere Möglichkeit ist, die Zuständigkeit explizit auf den Proxy-Agenten zu verlagern und dies bekannt zu geben, so daß im erzeugten Agenten die Referenz entsprechend eingetragen werden kann.

Anforderungen an den Eventmanager

Die Anforderungen sind ähnlich gelagert wie in Abschnitt II.4.2 .

- **zuverlässige Kommunikation**
  Der Request-Event muß garantiert beim Auftraggeber ankommen. Ebenso muß der Grant-Event garantiert bei den Consumern ankommen. Geht einer der beiden Events verloren, wird der Agent von der Location terminiert.

- **Mobilität des Auftraggebers**
  Dieser Punkt ist ähnlich dem vorigen. Wenn ein Request-Event auftritt, solange der Auftraggeber gerade migriert, muß der Event trotzdem korrekt ausgeliefert werden.

- **Absenderkennung**
  Zur Vermeidung von Agenten, die allen anderen jeglichen Request genehmigen, brauchen wir eine Absenderkennung im Event. Die Proxies aus der Verbesserung benötigen eine Sonderbehandlung.

Kosten

Basisalgorithmus

Für jeden Agenten des Auftraggebers wird pro Lebenszeitintervall ein Request-Event und ein Grant-Event über den Eventchannel versendet. Das ist sehr teuer, da es genaugenommen eine Kommunikation zwischen zwei Agenten ist, die lediglich ihren Aufenthaltsort nicht kennen. Das Weiterleiten dieser Events an jeden anderen Teilnehmer des Channels ist unnötig. Es ergeben sich folgende Kosten pro Lebenszeitintervall:

$$Kosten_{pOD,\ Basis} = \#Teilnehmer \times Kosten_{Event} \times 2 \qquad (II.13)$$

$Kosten_{Event}$ kann mit Formel *(II.11)* berechnet werden.

Direkte Variante

Wenn der Grant-Event mit direkter Nachrichtenkommunikation vom Auftraggeber zur betroffenen Location gesendet wird, wird der Eventchannel nur einmal belastet. Dafür entstehen nun Kosten für das Senden der Nachricht:

$$Kosten_{Nachricht} = \sum_{i=1}^{n} W_i \qquad (II.14)$$

mit

n	= Anzahl der Hops zwischen Auftraggeber und anfragender Location,
W_i	= $w(E_i)$, Kostenfunktion für Hop i .

Damit ergeben sich die Kosten für die zweite Variante zu

$$Kosten_{pOD,\ direkt} = \#Teilnehmer \times Kosten_{Event} + \#Teilnehmer_{Leben} \times Kosten_{Avg,\ Nachricht} \quad (II.15)$$

mit

#Teilnehmer	= Gesamtzahl Agenten in der Gruppe,
$\#Teilnehmer_{Leben}$	= Anzahl Agenten, die weiterleben dürfen,
$Kosten_{Avg,\ Nachricht}$	= durchschnittliche Kosten für das Senden einer Nachricht an einen beliebigen Empfänger. Diese sind niedriger als $Kosten_{Event}$.

Channelvariante

Besser sieht es aus, wenn der Grant-Event über den Eventchannel an alle Teilnehmer zugestellt wird. Der Request-Event der Location, die zuerst sendet, wird von den meisten anderen Locations empfangen, bevor sie ihren eigenen Request-Event absetzen. Je nachdem wie groß der Zeitraum vor Ende der Lebenszeit im Verhältnis zur Verzögerung der Übertragung eines Events an alle Teilnehmer gewählt ist, haben mehr oder weniger viele Locations ihren eigenen Request bereits verschickt, bevor der erste Request-Event bei ihnen eintrifft. Sei x die Anzahl der Locations, die einen Request-Event senden. Dann fallen noch folgende Kosten an:

$$Kosten_{pOD,\ Channel} = x \times Kosten_{Event} + Kosten_{Event} \approx 2 \times Kosten_{Event} \quad (II.16)$$

Als Reaktion auf den Request-Event braucht nur ein Grant-Event gesendet zu werden. Alle weiteren Request-Events für dieses Lebenszeitintervall können wie Duplikate des ersten Request-Events behandelt werden. Die Kosten liegen also deutlich unter denen des Basisalgorithmus.

Wenn man die Zusatzfunktionalität der direkten Variante außer acht läßt (das heißt $\#Teilnehmer = \#Teilnehmer_{Leben}$), ist die letzte Variante deutlich effizienter als die zweite. Soll aber für jeden einzelnen Teilnehmer bestimmt werden, ob er weiter leben darf, kann nur noch am spezifischen Einzelfall eine Entscheidung getroffen werden, ob die direkte Variante oder die Channelvariante besser ist, da dann das Verhältnis von überlebenden zu sterbenden Agenten den Ausschlag gibt.

Skalierbarkeit

Der Aufwand für den Basisalgorithmus skaliert linear mit Faktor 2 zur Gruppengröße. Die direkte Variante skaliert ebenfalls linear im Verhältnis zur Gruppengröße mit Faktor 2. Dafür bietet sie aber auch mehr Funktionalität. Auch der Aufwand für die Channelvariante wächst linear, allerdings mit einem Faktor knapp über 1.

Dies sind jeweils gemittelte Werte unter der Annahme, daß die durchschnittlichen Kosten für das Versenden eines Events ebenfalls linear mit der Gruppengröße wachsen.

II.4.4. Waisenverhinderung: Das Energiekonzept

Das Energiekonzept wurde bereits von Zepf in seiner Studienarbeit betrachtet [Zepf96]. Er kommt zu dem Schluß, daß es unpraktisch ist, wenn der Agent, und damit der Programmierer, selbst für die Energieversorgung verantwortlich ist. In diesem Abschnitt soll untersucht werden, ob bei Verwendung eines Eventmanagers diese Einschätzung beibehalten werden muß, oder ob sich eine elegante Lösung finden läßt. Das Energiekonzept ist insbesondere deshalb interessant, weil es sich auch gut für Accountingzwecke einsetzen läßt.

Prinzip

Jeder Agent wird beim Erzeugen mit einer gewissen Menge Energie "betankt". Für jede Aktion, die er ausführt, wird ihm von der Location Energie abgezogen. Auch für das einfache Existieren verbraucht er "Lebensenergie", die ihm von der Location in regelmäßigen Abständen abgezogen wird. Irgendwann kommt der Zeitpunkt, zu dem er keine oder nur noch sehr wenig Energie hat. Dann muß er sich an seinen Auftraggeber wenden und um Energie bitten. Wenn der Agent keine Energie mehr hat, wird er von der Location terminiert. Der Anstoß zum Auftanken kommt also vom Agenten. Der Status des Energietanks muß unter Umständen auch in anderen Programmteilen des Agenten berücksichtigt werden, damit bei niedrigem Füllstand keine Aktionen ausgeführt werden, die dem Agenten die restliche Energie entnehmen würden und damit seine Terminierung durch die Location einleiten. Mit der Ausführung solcher Aktionen muß unter Umständen bis zum nächsten Auftanken gewartet werden oder stattdessen eine "billigere" Aktion durchgeführt werden. Diese Eigenschaften machen die Programmierung eines "Energieagenten" kompliziert und damit fehleranfällig.

Ein Protokoll

Die Übertragung der Anfragen nach Energie und das Versenden von "Energiepäckchen" als Antwort kann vom Eventmanager übernommen werden. Wie bei allen bisher vorgestellten Algorithmen kann davon ausgegangen werden, daß für jede Aufgabe ein eigener Eventchannel existiert. Jeder Agent, der für diese Aufgabe arbeitet, gehört dieser Gruppe an. Der oder die Auftraggeber beantworten die Anfragen nach Energie mit Energie-Events, die die im Agenten vorhandene Energie wieder erhöhen.

Was bringt es uns?

Der große Vorteil dieses Verfahrens ist, daß wenn ein Agent keine Energie mehr hat, er automatisch ohne weiteres Zutun des Programmierers terminiert wird. Wenn man den Begriff "Energie" durch "Geldeinheiten" ersetzt, ist es zu einem Accountingsystem, bei dem für jede Leistung der Location bezahlt wird, nicht mehr weit. Für diesen Zweck werden aber vertrauenswürdige Locations und Kommunikationspfade benötigt. Für den Betreiber einer Location kann es schon reizvoll sein, dem Agenten das Geld aus der Tasche zu nehmen. Der Schutz von Agenten vor bösartigen Locations ist allerdings ein prinzipielles Problem von Agentensystemen und bisher nicht befriedigend gelöst.

Zum Auftanken wird eine Instanz benötigt, die den Agenten während seiner Lebenszeit immer wieder auftanken kann und dies auch darf. Das kann man durch einen Repräsentanten der Aufgabe, den Vater des Agenten oder den Group Coordinator einer Agentengruppe erledigen lassen. Wenn die "Tankstellen" strategisch geschickt plaziert sind, lassen sich so sogar Kommunikationskosten sparen. Eine Aufgabe kann beispielsweise in mehrere Teilaufgaben aufgeteilt werden, die jeweils einem Agenten übergeben werden, der wiederum Agenten zur Erledigung der Aufgabe erzeugt. Auch für jede (Teil)aufgabe wird dann ein eigener Eventchannel für die Verteilung der Energie-Events verwendet. Sinnvollerweise gibt es pro Gruppe nur einen Supplier von Energie-Events.

Beurteilung

Man sieht schnell, daß Events nicht besonders gut zur Energieverteilung geeignet sind. Das Konzept der Events ist dann vorteilhaft, wenn eine Information an beliebig viele, potentiell unbekannte Empfänger verteilt werden soll. Zur Verteilung der Energie ist es nicht wünschenswert, wenn jeder Agent dieselbe Menge an Energie bekommt. Das Versenden der Energie mit Events entspricht in der Realität einer Vervielfachung der ursprünglichen Energiemenge um die Anzahl der Agenten, die den Event empfangen. Übertragen auf die Idee des elektronischen Geldes käme dies einer Gelddruckerei gleich.

Darüberhinaus müssen mächtige Sicherheitsmechanismen eingeführt werden, die den Zugang zu dem Eventchannel auf dem die Energie verteilt wird, kontrollieren. Die Vorteile von Events kehren sich hier zum Nachteil um, denn der Sender des Events hat keine Kontrolle darüber, welche Agenten Energie von ihm erhalten. Er kennt die Empfänger nicht.

Für eine effiziente Energieverteilung ist ein hierarchisches System, dessen Struktur sich an die Gruppenstruktur der Agenten anlehnt und für die Verteilung Multicastmechanismen verwendet, besser geeignet.

II.4.5. Folgerungen

Wie zu Beginn dieses Kapitels bereits beschrieben bleibt als Hauptproblem aller Verfahren zur Waisenerkennung die "oberste Ebene", die Schnittstelle zwischen Anwendung und Agentensystem. Eine Instanz der Anwendung muß innerhalb des Agentensystems existieren, wenn man voraussetzt, daß sich ein Anwender mit mobilem Rechner, Modem und Telefonleitung in das System einwählt, Agenten absetzt und dann die Verbindung wieder beendet. Die Anwendungsinstanz darf dann natürlich nicht von der Waisenerkennung erfaßt werden, obwohl sie, solange die Verbindung zum mobilen Rechner nicht existiert, keine Lebensberechtigung im Sinne der hier vorgestellten Verfahren hat. Hier muß noch mehr Arbeit investiert werden.

II.5. Eigenschaften von Events

In diesem Kapitel beschäftigen wir uns mit der Frage, welche Eigenschaften ein Event besitzen muß. Zunächst wird untersucht, welche Eventarten in einem Eventsystem überhaupt auftreten können. Dann werden grundlegende Fragen der Speicherung und der Lebensdauer von Events besprochen und die Frage diskutiert, ob und welche Reihenfolgebeziehungen zwischen Events existieren müssen.

II.5.1. Eventarten

In einem verteilten Eventsystem lassen sich drei Grundarten von Events unterscheiden.

- **transiente Events**
  Für einige Aufgaben genügt es bereits, Events an alle **erreichbaren** Consumer weiterzuleiten. Consumer, die zum Beispiel wegen einer Netzpartitionierung oder Ausfall von Knoten nicht erreichbar sind, erhalten transiente Events nicht. Anwendung finden solche Events in der reinen Informationsverbreitung, bei der die Empfänger zustandslos sind, ein fehlender Event also nicht tragisch ist.

- **zuverlässige (reliable) Events**
  Diese Events werden garantiert an alle Consumer eines Channels weitergeleitet. Sollte ein Consumer nicht erreichbar sein, so werden sie vom Eventmanager bis zur Auslieferung aufbewahrt. Für Synchronisierungsaufgaben oder die Terminierung von Agenten ist diese Eventart unverzichtbar.

- **persistente Events**
  Persistente Events werden nicht nur, wie reliable Events, an alle Verbaucher ausgeliefert, sondern zusätzlich vom Eventmanager für einen späteren Abruf gespeichert.

Transiente Events sind sehr einfach zu realisieren. Auf Fehler braucht keine Rücksicht genommen zu werden. Hingegen ist die Realisierung von zuverlässigen und persistenten Events schwieriger und in grundsätzlichen Fragen sehr ähnlich. In den folgenden Abschnitten betrachten wir das genauer.

II.5.2. Speicherung von Events

Zuverlässige, und vor allem persistente, Events müssen unter Umständen sehr lange vom Eventmanager aufbewahrt werden. Der Speicherplatzverbrauch dafür kann unter Umständen immens sein. Die Speicherung aller Events ist aber nicht immer notwendig, da ein Event für einen Consumer meist nur in einem begrenzten Zeitraum nützlich ist.

Beispielsweise ist es nicht nötig einen Terminierungsevent für einen Agenten nach der Entfernung des Agenten aus dem System noch zu speichern, da der Agent nicht mehr existiert und mit Sicherheit den Terminierungsevent nicht mehr empfangen kann.

Zunächst untersuchen wir, wo persistente und zuverlässige Events sinnvoll gespeichert werden. Dann betrachten wir Möglichkeiten, die Lebensdauer von Events einzuschränken, ohne Funktionalität oder Informationen zu verlieren.

Der Speicherort von Events

In diesem Abschnitt werden nur zuverlässige und persistente Events betrachtet. Transiente Events brauchen per defitionem nicht gespeichert werden und bedürfen auch keiner Fehlerbehandlung.

Es gibt drei Gründe, Events innerhalb des Eventmanagementsystems zumindest kurzfristig zu speichern.

- Fehlerfälle müssen berücksichtigt werden. Das Kommunikationssubsystem muß sicherstellen, daß die Events an die jeweils nächste Station sicher weitergeleitet werden.

- Zuverlässige Events müssen solange im System verfügbar bleiben, bis sie an jeden Empfänger ausgeliefert wurden. Der Schwerpunkt der Aufgabe liegt hier nicht beim Speichern an sich, sondern in der Feststellung, wann der Event jeden Empfänger erreicht hat und eventuell gespeicherte Kopien des Events gelöscht werden können.

- Persistente Events müssen aufgrund ihrer Definition auf stabilem Speicher abgelegt werden. Es muß untersucht werden, wo dies geschehen soll.

Zuverlässige Events

Events werden, um die Kosten möglichst gering zu halten, vom Eventmanager über einen spannenden Baum versendet. Wie der Baum erzeugt und gewartet wird, betrachten wir in Teil III dieser Arbeit. Für die weiteren Ausführungen genügt es zu wissen, daß ein Baum existiert, der alle Consumer eines Events verbindet. Sei der Sender des Events die Wurzel des Baums. Für zuverlässige Events genügt es für jeden Knoten festzustellen, daß der gesendete Event jeden Knoten in jedem Unterbaum des sendenden Knotens erreicht hat.

Die Bedingung läßt sich leicht rekursiv bestimmen. Ein Event wird von jedem Knoten an alle Nachbarknoten außer dem Sender weiterverteilt. Wenn der Event bei einem Knoten eingeht, wird dieser lokal verarbeitet (zum Beispiel Speicherung auf stabilem Speicher, Prüfung von Reihenfolgebedingungen, ...) und der Empfang des Events beim Sender bestätigt. Dann versendet er den Event weiter in die Unterbäume. Zu beachten ist, daß wir zwischen Empfang und Auslieferung eines Events unterscheiden! Die Auslieferung eines empfangenen Events an die Consumer kann aufgrund von möglichen Reihenfolgebedingungen erst später erfolgen.

Ein Event bleibt solange auf dem Wurzelknoten eines Unterbaums gespeichert, bis der Wurzelknoten von jedem seiner Kinder aus dem Unterbaum eine Empfangsbestätigung

erhalten hat. Sendet ein Knoten nach lokaler Verarbeitung eine Bestätigung für einen Event, so ist garantiert, daß der Event nicht nochmals gesendet werden muß.

Persistente Events

Persistente Events müssen irgendwo im Eventmanagementsystem gespeichert werden. Zusätzlich zur gesicherten Übertragung, wie wir sie im vorigen Abschnitt beschrieben haben. Prinzipiell bieten sich dazu zwei Möglichkeiten an.

- **stabiler Speicher beim Sender**
  Die einfachste Methode persistente Events zu erhalten ist, sie vor dem Senden lokal auf stabilem Speicher in einem Protokoll abzulegen. Der sendende Rechner ist somit dafür verantwortlich, daß persistente Events verfügbar bleiben und auf Anforderung nochmals versendet werden können. Diese Bedingung ist für mobile Rechner, die nur zeitweise am Netz sind, nicht unbedingt praktikabel.

- **Kernsystem**
  Eine Verfeinerung der stabilen Speicher ist ein vertrauenswürdiges Kernsystem bestehend aus mehreren Rechnern. Das Kernsystem ist im Idealfall so verteilt, daß bei einer Netzwerkpartitionierung pro Partition noch ein Rechner des Kernsystems verfügbar ist. Knoten des Kernsystems sind Consumer in allen (oder auch nur bestimmten) Eventchannels und protokollieren sämtliche versendeten persistenten Events mit. Sollte ein Event von einem lokalen Eventmanager nochmals angefordert werden und ist der Event nicht mehr beim Sender gespeichert oder der Sender nicht verfügbar, so kann der Event aus dem Archiv des Kernsystems geholt werden. Die Einschränkung auf bestimmte Eventchannels, bei denen Zuverlässigkeit wichtig ist, ist notwendig, weil die Anzahl versendeter Events in einem Eventchannel je nach Anwendung erheblich sein kann. OrbixTalk bietet mit dem MessageStore eine ähnliche Funktionalität [Orbix96].

Es ist leicht einzusehen, daß das Protokoll, beziehungsweise das Archiv des Kernsystems, nicht ewig wachsen kann und daher Mechanismen vorgesehen werden müssen, mit denen das Protokoll "ausgedünnt", das heißt von Events befreit werden kann, die mit Sicherheit nicht mehr angefordert werden.

II.5.3. Lebensdauer von Events

Neben der Frage *wo* all die Events gespeichert werden, ist die Frage *wie lange* die Events gespeichert werden noch viel interessanter. Aus Sicht des Eventmanagers müssen die Events so lange gespeichert werden, wie sie eventuell nochmal angefordert werden könnten. Dafür aber eine konkrete Zeit festzulegen ist leider nicht möglich, da sie von vielen Einflüssen abhängt (Rechenzeit, Migrationszeit, ...). Wieder einmal ist hier die Mithilfe der Anwendung (und damit des Programmierers dieser Anwendung) nötig.

Hinzu kommt, daß nicht jeder Event gleich lang gespeichert werden muß. Es gibt einerseits Events, die nur in einem sehr kurzen Zeitrahmen sinnvoll sind (zum Beispiel Uhrenticks) und andererseits Events, die eine sehr lange Gültigkeit haben (zum Beispiel Terminierung). Zur Lösung dieses Problems kann man den Events eine Lebensdauer geben, nach deren Ablauf sie nicht mehr gespeichert werden. Versendet werden sie natürlich trotzdem noch an jeden interessierten Consumer, auch wenn ihre Lebensdauer bereits abgelaufen ist. Solche Events können dann wie transiente Events behandelt werden.

In den bisher betrachteten Beispielen, in denen der Eventmanager verwendet wurde, gibt es mehrere Events, die in mehr oder weniger regelmäßigen Intervallen gesendet wurden (zum Beispiel die "x is alive"-Events). Der nachfolgende Event beendet die Gültigkeit des vorhergehenden Events. Dies läßt sich zum Beispiel über ein Statusfeld des Events spezifizieren (Event *b* löst Event *a* ab). Die Verwendung solcher Events kann auch für die Optimierung des Routings von Events innerhalb des Eventmanagers sinnvoll sein ("Update-Events" [Baumann97b]).

Terminierungsevents haben im Prinzip eine unendliche Gültigkeitsdauer und müssen daher auch unendlich lang aufbewahrt werden. Diese Forderung läßt sich aber sinnvoll abschwächen. Ist der Terminierungsevent vom betroffenen Agenten verarbeitet, braucht er nicht mehr gespeichert werden, denn er wird mit Sicherheit nicht mehr angefordert, da der betroffene Agent bereits tot ist. Wenn wir darüberhinaus annehmen, daß jeder Agent einer Terminierungsgruppe angehört, die über einen gemeinsamen Eventchannel kommuniziert, so können Terminierungsevents zu dieser Gruppe gelöscht werden, sowie die Gruppe aufgelöst und der Eventchannel gelöscht wird.

Allgemein läßt sich sagen, daß wenn der Eventchannel geschlossen wird oder keine Consumer beziehungsweise Supplier mehr hat, sämtliche gespeicherten Events zu diesem Eventchannel gelöscht werden können.

II.5.4. Reihenfolge von Events

Wir beschäftigen uns nun mit der Frage, für welche Einsatzbereiche des Eventmanagers eine Reihenfolgebeziehung zwischen verschiedenen Events überhaupt nötig und sinnvoll ist.

Terminierung und Waisenerkennung

Es ist leicht einzusehen, daß ausschließlich für Terminierungsevents Reihenfolgebeziehungen weder nötig, noch sinnvoll sind: Es wird nur ein einziger Event zur Terminierung eines Agenten versendet. Der Eventmanager muß sicherstellen, daß dieser Terminierungsevent beim Agenten eintrifft.

Betrachten wir die vorgestellten Protokolle zur Waisenerkennung, so fallen einige Eigenschaften auf:

- Bei der aktiven Waisenerkennung werden im Extremfall nur Statusmeldungen ("x is alive") versendet. In welcher Reihenfolge verschiedene Statusmeldungen bei verschiedenen Consumern eingehen, ist für die Funktion des Protokolls irrelevant.

- Bei der passiven Waisenerkennung gibt es Request-Events und als Reaktion darauf Grant-Events. Logischerweise kann ein Grant erst nach dem Empfang des Requests von dem zuständigen Objekt erzeugt werden. Auch hier haben wir kein Reihenfolgeproblem, solange sichergestellt ist, daß sich nur ein Objekt zuständig fühlt, die Requests zu behandeln.

Aus den in dieser Arbeit untersuchten Aufgabenbereichen Terminierung und Waisenerkennung lassen sich somit keine Bedingungen an die Auslieferungsreihenfolge von Events stellen.

Synchronisierung

Da der Eventmanager auch für Synchronisierungszwecke zwischen verschiedenen Agenten oder Prozessen des Agentensystems eingesetzt werden soll, benötigen wir stärkere Bedingungen für die Reihenfolge von Events.

Als Minimalanforderung für Synchronisierung ergeben sich schnell zuverlässige Events in kausaler Ordnung [Baumann97b]. Eine genauere Beschreibung der Hintergründe findet sich in der parallel laufenden Studienarbeit von Szasz [Szasz97].

II.6. Mobilität von Objekten

Der Eventmanager soll es erlauben, mobilen Agenten bei minimalem Aufwand Events zuzustellen. Diese Forderung hat starke Auswirkungen auf die Konstruktion des Eventmanagers. Viele Verfahren für das Verteilen von Nachrichten über große, weiträumige Netzwerke verwenden (minimal) spannende Bäume (MSB) zwischen den Knoten, auf denen Objekte einer Aufgabe agieren. Die Nachrichten werden entlang der Kanten dieser Bäume zwischen den Teilnehmerknoten verteilt. Der wohl beste und bekannteste verteilte Algorithmus zur Bestimmung eines annähernd minimal spannenden Baums über einer Teilmenge von Knoten eines Netzwerks stammt von Wall [Wall80].

Knoten kann man in diesem Zusammenhang immer mit Rechner gleichsetzen, da die Verteilung der Events zwischen verschiedenen Rechnern in eventuell unterschiedlichen Subnetzen im Verhältnis zur lokalen Eventverteilung innerhalb des Rechners sehr teuer ist. Zur Vereinfachung gehen wir, wie im ersten Teil beschrieben, davon aus, daß pro Rechner ein Prozeß, der Eventdämon, für die Kommunikation mit anderen Eventdämonen, sowie eventuell anderen Teilen des Eventmanagers (zum Beispiel in Routern) zuständig ist.

Spannende Bäume haben die geringsten Kommunikationskosten, um eine Nachricht zwischen allen Knoten zu verteilen. Bei n Knoten wird die Nachricht n-1-mal übertragen. Wenn sich die Struktur des MSB an der Topologie des Netzwerks orientiert, ist sichergestellt, daß auch jeder Link nur einmal mit dieser Nachricht belastet wird.

Das Problem der Bestimmung eines minimal spannenden Baums über einer Untermenge von Knoten eines Netzwerks unter der Prämisse, daß einige Knoten im MSB sein müssen und beliebig viele andere Knoten des Netzwerks zum Erstellen des MSB mitverwendet werden dürfen, ist leider NP-vollständig. Es handelt sich dabei um das "Steiner-Tree-Problem" [Wall80]. Walls Algorithmus verwendet Heuristiken um einen spannenden Baum zu erzeugen, der dem optimalen Steiner-Tree im schlechtesten Fall ähnlich ist. Für "normale" Computer-Netzwerke berechnet Walls Algorithmus immer den optimalen MSB.

Der Algorithmus von Wall erzeugt zwar sehr gute, das heißt billige, MSBs, geht allerdings davon aus, daß sich die Gruppen von Knoten, die sich für einen Event interessieren, nicht verändern. Kommt ein Knoten hinzu oder verläßt einer die Gruppe, so muß die Berechnung des MSBs komplett neu durchgeführt werden. Das kann teuer werden.

Migriert der Agent von einem Knoten auf einen anderen, so muß der MSB ebenfalls neu berechnet werden, wenn sich dadurch die Gruppenzugehörigkeit der Knoten ändert. Dieser Fall kommt in einem System mobiler Agenten sehr häufig vor. Daher ist es nicht akzeptabel den MSB bei jeder Migration neu zu berechnen. Es ist allerdings auch nicht unbedingt nötig, einen minimal spannenden Baum für die Verteilung der Events zu haben, obwohl dafür die Kommunikationskosten natürlich optimal gering wären.

In Kapitel III.2.2 stellen wir einen verteilten Algorithmus zur Verwaltung spannender Bäume vor, der auch für migrierende Objekte geeignet ist [Belkeir89].

II.6.1. Mobile Consumer

Welche Auswirkungen hat die Mobilität eines Consumers auf das Management von Events? Zunächst verkompliziert sie das Verfahren der Verteilung der Events, wenn wir garantieren, daß jeder Event ausgeliefert wird. Eine Eigenschaft, die wir zum Beispiel für Terminierungsevents voraussetzen wollen.

Die Laufzeit eines Events von der Quelle zu allen Consumern kann, bedingt durch die Verteilung im Netzwerk entlang der Kanten des spannenden Baums, unterschiedlich lang sein. Wenn der Agent genau in dieser Zeitlücke von einem Ort, der den Event noch nicht erhalten hat, zu einem Ort migriert, der den Event bereits verarbeitet hat, so ist er zum Zeitpunkt der Auslieferung eines Events an keinem der beiden Orte und erhält den Event auch nicht. Abbildung II.4 illustriert diesen Fall. Wir brauchen zusätzliche Maßnahmen, um solche Schlupflöcher zu verhindern.

Ein Lösungsansatz ...

Ein naheliegender Ansatz funktioniert folgendermaßen: Zu jedem Eventchannel existiert eine eindeutige Numerierung der Events, die vom Eventmanager verwaltet wird. Events werden bei allen Consumern in genau dieser Reihenfolge ausgeliefert.

Wenn der Agent migrieren will, so hat er alle Events erhalten, die bei seinem bisherigen Ort eingetroffen sind. Während der Migration kann er natürlich keine Events empfangen. Nach der Migration stellt der Zielort fest, bis zu welcher Eventnummer der Agent Events aus der entsprechenden Gruppe bekommen hat und liefert die noch fehlenden Events aus.

... und die Folgen

Wenn der empfangende Ort bisher bereits Consumer dieses Channels hatte, ist bekannt, bis zu welcher Eventnummer der Ort bereits Events verarbeitet hat. Die fehlenden Events müssen beim Eventmanager angefordert werden. Eventuell kann auch der Ort, beziehungsweise der lokale Eventmanager, einen lokalen "Cache" von zurückliegenden Events verwalten.

Aufwendiger wird es, wenn der Ort bisher noch keine Consumer des entsprechenden Channels hatte, da er bisher keine Events dieses Channels empfangen hat. Es muß die Kommunikationsstruktur des Eventmanagers geändert werden und eventuell der spannende Baum umgebaut werden. Das trifft auch auf den Ort zu, den der Agent verlassen hat. Sollte dort kein weiterer Consumer des Channels vorhanden sein, so muß auch dort die Kommunikationsstruktur des Eventmanagers umgebaut werden, da dieser Knoten nun keine Events des Channels mehr empfangen braucht.

Eine Migration bewirkt unter Umständen zwei Änderungen in der Kommunikationsstruktur des Eventmanagers.

Die vorgestellte Methode verlangt vom Eventmanager eine eindeutige, geordnete Numerierung der Events. Solange garantiert nur ein Supplier sendet, ist das kein Problem. Die Eventnummer wird lokal bei diesem Supplier verwaltet und einfach mit jedem erzeugten Event erhöht. Wie sieht das aber bei mehreren Suppliern aus? Dann muß die eindeutige,

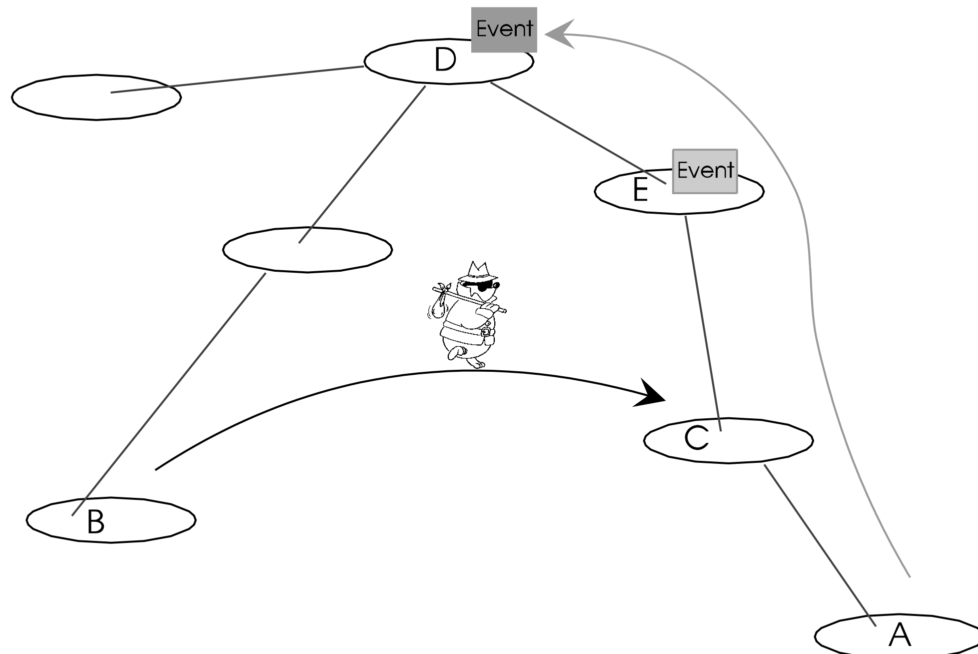


Abb. II.4: Der Terminierungsevent wird von Knoten A verschickt. Der zwischen Knoten B und C migrierende Agent bekommt den Event nicht zu sehen, da er bei Eintreffen des Agenten in Knoten C bereits bearbeitet ist. Der Terminierungsevent hat soeben Knoten D erreicht und wird von Knoten E noch bearbeitet.

globale Numerierung zwischen den Suppliern synchronisiert werden. Die einfachste Möglichkeit, bei einer zentralen Komponente des Eventmanagers die Nummern zu verwalten, ist aufgrund der Vorgabe, daß kein Single-Point-Of-Failure existieren darf, nicht ohne weiteres akzeptabel. Darüberhinaus ist auch die Laufzeit einer Nachricht in einem weit verteilten System nicht zu vernachlässigen. Es kann einige Zeit dauern bis ein Event bei der Zentrale eingegangen, global numeriert und wieder versendet ist.

Einen solchen Algorithmus stellen wir später in Kapitel III.2.6 vor [Kaashoek91].

II.6.2. Mobile Supplier

Mobile Supplier stellen kein besonderes Problem dar. Sie übergeben ihre zu sendenden Events an den lokalen Eventmanager, der sich um die korrekte und sichere Zustellung der Events kümmert. Der Einwand, daß der Eventmanager zurückliegende Events vom Supplier erneut anfordern könnte, ist nicht sonderlich sinnvoll, da dann der Supplier für die Aufbewahrung seiner alten Events zuständig wäre und bei einer Migration auch mitnehmen müßte. Das ist bestenfalls dann akzeptabel, wenn ein Event auf Anforderung neu *berechnet und erzeugt* werden könnte. Dies trifft aber für die wenigsten Einsatzzwecke zu.

Schwieriger ist die Einhaltung der korrekten Reihenfolge von Events, die vom selben Supplier von unterschiedlichen Orten auf unterschiedlichen Rechnern gesendet wurden. Zu diesem Zweck muß ein beliebiger Supplier jederzeit eindeutig für den Eventmanager identifizierbar sein.

II.6.3. Binden von Objektreferenzen

Der Eventmanager muß die Objekte referenzieren können, die sich bei einen Eventchannel registrieren lassen. In einem lokalen System ist das kein Problem. Am einfachsten ist die Referenzierung über einen simplen Zeiger auf einen gemeinsamen Speicherbereich. Da unsere Objekte, die Agenten, potentiell zwischen verschiedenen Rechnern migrieren können, benötigt jedes Objekt eine systemweit eindeutige Identifikation, unter der es sich beim Eventmanager anmelden kann, die sich auch nicht ändert, wenn das Objekt migriert.

Berechnung einer Identifikation

Ein einfacher (aber leider nicht praktikabler) Ansatz arbeitet folgendermaßen. Der Eventmanager verwaltet keine Referenzen auf die Objekte selbst, sondern berechnet eine eindeutige Identifikation zu dem Objekt (als Hashcode, CRC, ...). Meldet sich ein Objekt nach erfolgter Migration beim lokalen Eventmanager der Zielengine an, erkennt dieser anhand der neu berechneten Identifikation um welches Objekt es sich handelt.

Das Verfahren hat einen kleinen Nachteil, der es für unsere Zwecke unbrauchbar macht: Der Status eines Agenten ändert sich üblicherweise im Laufe seines Lebens, und damit auch seine Identifikation. So geht es also nicht.

Ein Keks vom Krümelmonster

Ein Objekt, das den Eventmanager verwenden möchte, muß sich beim ersten Mal anmelden. Als Ergebnis der Anmeldung erhält das Objekt vom Eventmanager ein weiteres Objekt: ein Cookie, über das alle weiteren Aktionen des Eventmanagers ausgelöst werden können. Die Verwendung eines Cookies hat für den Eventmanager wertvolle Vorteile.

- **eindeutige Identifikation und lokale Erzeugung**
  Der Eventmanager kann im Cookie Informationen ablegen, die das Objekt eindeutig beschreiben.
  Der lokale Eventmanager, bei dem sich das Objekt das erste Mal anmeldet, bestimmt lokal eine global eindeutige Identifikation für das Objekt. Die Identifikation setzt sich zusammen aus der Netzwerkadresse der Engine, der lokalen Uhrzeit und einer laufenden Nummer. Die Netzwerkadresse einer Engine ist immer eindeutig (IP-Adresse und Portnummer). Die Uhrzeit vermeidet doppelt vergebene Numerierungen nach einem Absturz einer Engine. Der Wertebereich der laufenden Nummer kann sehr klein sein, denn ihr Hauptzweck ist die Vermeidung von Duplikaten bei Registrierungen innerhalb der Ausleseungenauigkeit der internen Rechneruhr.
  Zur Bestimmung der minimalen Größe eines Cookies dieser Methode werfen wir kurz

einen Seitenblick auf die Implementierung. Eine IP-Adresse ist 32-bit lang (In der nächsten Version von IP (v6) wird sie etwas länger sein: 128 bit). Die Portnummer der Engine ist ein 16-bit Wert. Die Uhrzeit nach UTC-Konvention wird in Java in einem 64-bit Wert gespeichert. Nehmen wir für die laufende Nummer nun noch einen 8-bit Wert an, dann sind für die global eindeutige Identifikation eines beliebigen Objekts 120 Bit notwendig. Das sind 15 Byte, um die ein migrierender Agent größer wird, wenn er ein Cookie dabei hat. In Anbetracht der Genauigkeit der Identifikation ist dieser Wert vertretbar.

- **Statusinformationen**
  In einem Cookie kann der Eventmanager aber nicht nur die Identifikation des Objekts speichern, sondern auch noch Statusinformationen ablegen. Zum Beispiel Referenzen auf Eventchannels, denen der Agent angehört und die ID des jeweils letzten empfangenen Events.

Schöne, böse Welt

Leider hat auch dieses Verfahren einen kleinen Haken. Angenommen, ein Agent kopiert einem anderen Agenten das Cookie. Der zweite Agent ist für den Eventmanager nicht vom Original unterscheidbar, da beide dasselbe Cookie zur Identifikation verwenden. Das kann vorteilhaft sein (zum Beispiel für Proxies der passiven Waisenerkennung), ist aber vom Ideal der eindeutigen Identifikation eines beliebigen Objekts meilenweit entfernt.

Weiterhin kann sich ein Agent jederzeit beim Eventmanager ein neues Cookie holen und auch parallel zum alten einsetzen. Dem Eventmanager werden damit von einem Objekt viele Objekte simuliert.

Agentennamen

Aufgrund dieser Nachteile betrachten wir nun doch die Verwendung von Agentennamen als Identifikation. Damit wird zwar die Verwendbarkeit des Eventmanagers auf das Agentensystem MOLE beschränkt, die beschriebenen Nachteile der Cookies können dann aber nicht mehr auftreten.

Mobile Objekte sind in MOLE immer Agenten. Ein Agent besitzt immer einen eindeutigen Namen, der ihm von der Location beim Erzeugen zugewiesen wird. Der Agent kann seinen Namen nicht ändern und ist damit immer eindeutig durch den Eventmanager identifizierbar.

Anders sieht das mit Objekten des MOLE-Kerns aus, die den Eventmanager für systeminterne Verwaltungsmechanismen verwenden wollen (zum Beispiel Locations). Diese Objekte sind keine Agenten und haben demzufolge auch keinen Agentennamen. Allerdings läßt sich dieses Problem elegant unter Verwendung von Systemagenten lösen. Ein Systemagent hat in MOLE dieselben Rechte wie systeminterne Objekte. Systemagenten sind nicht mobil und können nur von autorisierten Personen beziehungsweise vom System selbst in das Agentensystem eingebracht werden. Damit stellt dieser Ansatz auch kein Sicherheits-

loch dar. Der Systemmonitor "Moleview" [Beck96] geht genau diesen Weg zur Übertragung von Systeminformationen einer Location an mehrere entfernte Clientprogramme, indem bei Bedarf ein Systemagent erzeugt wird ("MoleviewAgent"), der die Verwaltung der Clients übernimmt und aus dem MOLE-Kern aufgerufen wird. Statt des eigentlichen Objekts aus dem MOLE-Kern meldet sich der Systemagent beim Eventmanager an und leitet die Events an das zugehörige Objekt weiter.

Bei dieser Variante ist zwar ein zusätzlicher Methodenaufruf nötig, um einen Event auszuliefern, der Systemagent braucht selber keinen Thread zu besitzen und belastet somit das Gesamtsystem nur geringfügig.

Fazit

Die lokale Referenzierung von Objekten durch den Eventmanager stellt kein Problem dar. Schwierig wird es erst durch die Mobilität der Agenten. Der vorgestellte Ansatz der "Cookies" ist für einen universell einsetzbaren Eventmanager sehr gut geeignet, da er im Cookie außer der eindeutigen Referenz auch noch interne Verwaltungsinformationen ablegen kann, die ohne weitere Probleme bei einer Migration vom Agenten selbstständig mitgenommen werden. Schwer wiegen allerdings Sicherheitsbedenken.

Die Referenzierung über den Agentennamen ist eine elegante Methode die Identifizierungsproblematik zu lösen, schränkt aber die Anwendung des Eventmanagers zunächst auf MOLE ein.

Optimal aus Sicht der Sicherheit und der Eleganz ist eine Kombination der beiden Verfahren. Im Cookie wird zusätzlich der Name des Agenten gespeichert und bei Aufrufen des Eventmanagers überprüft. Somit können Verwaltungsinformationen beim Agenten abgelegt werden, wenn das notwendig ist, und der Agent nimmt diese sogar ohne weiteres bei einer Migration zum Zielsystem mit. Bei einer Neuanmeldung am Zielort genügt dann ein einziger Methodenaufruf des Eventmanagers, um den Agenten wieder bei allen entsprechenden Eventchannels anzumelden.

II.7. Einfügen neuer Knoten

In allen bisher dargestellten Szenarien gehen wir davon aus, daß sich entweder keine neuen Hosts an den spannenden Bäumen beteiligen, oder daß sich im LAN ein Rechner befindet, bei dem sich neu hinzukommende Hosts anmelden können. Wie kann man diese neuen Systeme in das laufende System einbinden?

II.7.1. Wo liegt das Problem?

Spezifizieren wir zunächst das Problem genauer in Hinblick auf MOLE. Eine Engine des Agentensystems wird auf einem Rechner gestartet. Bei Agentenmigration und Kommunikation über Messages ist das Ziel beziehungsweise der Empfänger bereits beim Senden bekannt. Daher sind in diesem Bereich auch keine weiteren Maßnahmen auf Systemebene notwendig, um die Kommunikation zu ermöglichen. Die Ziele werden von Agenten vorgegeben und entweder funktioniert die Kommunikation oder sie funktioniert nicht.

Bei Events ist das vollkommen anders. Die neu gestartete Engine muß in das Eventmanagementsystem eingebunden werden. Das bedeutet, daß der Eventmanager über neu hinzugekommene Engines informiert werden muß, um diesen die für sie bestimmten Events zustellen zu können. Im Gegensatz zur Messagekommunikation hat ein Event aber kein explizites Ziel. Stattdessen verwaltet der Eventmanager die interessierten Engines des Agentensystems. Die Frage ist nun: Wie teilt eine neu gestartete Engine dem Eventmanager mit, daß sie jetzt aktiv ist und möglicherweise Events empfangen kann?

II.7.2. Lösungsansätze

Betrachten wir noch einmal das Modell des Eventmanagers aus der Einleitung. Pro Rechner ist normalerweise ein Dämonprozeß, der Eventdämon, für die Kommunikation des Eventmanagers zuständig. Alle Engines, die auf dem Rechner gestartet werden, versenden und erhalten ihre Events über ihren lokalen Eventmanager von einem Eventdämonen. Die Aufgabe reduziert sich auf die Fragen, wie der Dämon Verbindung zum restlichen System aufnimmt und wie der lokale Eventmanager in der Engine Verbindung zum Dämonen aufnimmt. Für beide Fragen kommen prinzipiell ähnliche Ansätze in Frage.

Statische Methoden

Die folgenden zwei Möglichkeiten arbeiten mit fest definierten Beziehungen zwischen den Dämonen.

Kernsystem

Dieser Ansatz geht davon aus, daß es eine Anzahl Rechner gibt, auf denen das Agentensystem ständig aktiv ist. Damit sind dort auch Instanzen des Eventmanagers aktiv. Der Dämon teilt einem anderen Dämon im Kernsystem mit, daß er an der Eventverteilung teil-

nehmen möchte. Diese Information wird vom Kernsystem an alle derzeit aktiven Dämonen weitergeleitet (zum Beispiel als systeminterner Event) und daraufhin die Kommunikationsstruktur des Eventmanagers auf den neuen Dämon ausgeweitet. Wir gehen davon aus, daß Eventdämonen nicht sehr häufig neu gestartet werden. Daher ist es zulässig, daß die Ausweitung der Kommunikationsstruktur einige Zeit dauern kann und unter Umständen Langstreckenübertragungen enthalten darf.

Graphenmethode

Dieser Ansatz ist eine Verallgemeinerung der vorigen Idee. Ein Dämon kennt eine Reihe von Rechnern auf denen normalerweise Eventdämonen aktiv sind. Die Idee hierbei ist, daß der Dämon die Einträge dieser Liste nun der Reihe nach abarbeitet und jeweils versucht den entsprechenden entfernten Dämon zu erreichen. Wenn man die Einträge der Liste lokal jeweils so ordnet, daß zunächst schnell erreichbare Rechner abgefragt werden und als Rückfalebene Rechner des Kernsystems, läßt sich unter Umständen eine schnellere Integration in das System erreichen.

Beide Ansätze haben das Problem, daß die Liste mit den Rechnernamen regelmäßig gewartet und aktualisiert werden muß.

> *Zumindest für die Adressen der Rechner des Kernsystems läßt sich die Aktualisierung organisatorisch an zentraler Stelle zusammenfassen. Die aktuelle Liste kann zum Beispiel auf dem WWW-Server der Fakultät Informatik abgelegt werden. Von dort kann sie während oder nach der Installation des Eventmanagers via HTTP geholt werden.*

Internet-Multicast

Bei dieser Variante kennt der hinzukommende Dämon keinen anderen Rechner der am Eventmanagement teilnimmt, und muß daher den Eventmanager suchen.

Der Dämon verwendet IGMP (siehe Kapitel III.2.1), um seinen Host als Mitglied in einer bestimmten Multicastgruppe anzumelden. Jeder Eventdämon ist Mitglied dieser Gruppe. Der neu gestartete Eventdämon sucht nun den "am nächsten" liegenden Eventdämon, indem er *join-request* in die Gruppe sendet. Die Lebenszeit (TTL) der *join-requests* ist begrenzt. Der erste Request hat einen Hop-Count von 1, daß heißt der Request wird nur im lokalen Netz empfangen. Meldet sich innerhalb eines Timeouts kein Eventdämon, so muß der neue Eventdämon davon ausgehen, daß er der erste in diesem Subnetz ist. Er sendet einen weiteren *join-request* mit um eins erhöhtem Hop-Count. Nun werden auch die direkt benachbarten Sub-Netze in die Suche miteinbezogen. Der Hop-Count wird solange erhöht, bis sich ein anderer Eventdämon meldet.

Bewertung

Die zuletzt beschriebene Variante verwendet elegant die Möglichkeiten von Multicasts zur Lokalisierung von Ressourcen. Allerdings ist MOLE zum heutigen Zeitpunkt noch nicht sehr weit verbreitet und ein in Amerika gestarteter Eventdämon benötigt viel Zeit und sehr

viel Netzwerkkapazität bis er einen Rechner mit einem aktivem Eventdämon (zum Beispiel in Stuttgart) findet. Diese Methode sollte erst angewendet werden, wenn MOLE weitverbreitet ist und ein aktiver Eventdämon mit wenigen Hops erreicht werden kann.

Die statischen Methoden sind zwar nicht ganz so elegant, aber für die heutige Situation besser geeignet, da sie direkt auf Rechner verweisen, auf denen mit hoher Wahrscheinlichkeit ein Eventdämon aktiv ist. Aus diesem Grund erscheint uns die Graphenmethode zum derzeitigen Zeitpunkt besser geeignet.

II.7.3. ... und wie findet ein lokaler Eventmanager seinen Dämon?

Der lokale Eventmanager einer MOLE-Engine kann beim Starten drei Situationen vorfinden.

- **Eventdämon lokal vorhanden**
  Auf dem Host ist bereits ein Eventdämon aktiv. Dann gibt es keine weiteren Probleme. Der lokale Eventmanager öffnet eine Verbindung zum Dämon und beginnt mit der Kommunikation.

- **kein Eventdämon lokal vorhanden, aber Starten möglich**
  Der lokale Eventmanager startet den Eventdämon, der sich nun um die Ankoppelung an den Eventmanager, wie im vorigen Abschnitt beschrieben, kümmern muß.

- **kein Eventdämon lokal vorhanden und Starten nicht möglich**
  Dieser Fall tritt ein, wenn entweder der Programmcode des Eventdämons lokal nicht lauffähig ist, oder der Rechner nicht aufwendig in den Eventmanager eingebunden werden soll, da er bald wieder vom Netz getrennt wird (wie es zum Beispiel für PDAs wahrscheinlich ist). In unserem Modell ist vorgesehen, daß sich der lokale Eventmanager an einen entfernten Eventdämon "ankoppelt". Auch hier bietet sich eine statische Konfiguration der Suche an. Es kann sogar dieselbe Liste verwendet werden, die im vorigen Abschnitt für die Suche des Eventdämons verwendet wird. Eventuell sollte geprüft werden, welcher der dort aufgeführten Rechner am nächsten zum aktuellen Standort des PDAs liegt.

Mit den in diesem Kapitel vorgestellten Methoden ist sichergestellt, daß eine neue MOLE-Engine schnell Zugriff auf den Eventmanager erhält.

II.8. Fehlerbehandlung

Dieses Kapitel behandelt übliche Fehler, die bei Multicasts auftreten können, und mögliche Lösungsansätze, diese Fehler zu verhindern, beziehungsweise zu reparieren. So gut wie alle Algorithmen für Multicasts in Weitverkehrsnetzen verwenden spannende Bäume zur Übertragung der Nachrichten. Daher beschränken wir uns in diesem Kapitel auf Fehlerfälle, die in diesen Bäumen auftreten können.

II.8.1. Ausfall einer Übertragungsstrecke ("Link")

Durch den Versuch die Kosten eines spannenden Baums möglichst gering zu halten, ergibt sich ein Baum, dessen Kanten (im allgemeinen Fall) auf den schnellsten Links des Netzwerks liegen. Ein "Link" ist eine Übertragungsstrecke zwischen zwei lokalen Eventmanagern, die auch über mehrere Router verlaufen darf. Wir betrachten nun den Fall, daß ein Link mitten in einer Übertragung zusammenbricht (das ist wohl der "worst case"). Grundlage ist das Internet-Protokoll (IP) in der Version 4.

Ausweichen möglich

Jeder gesendete Event wird prinzipiell den schnellsten Weg von Sender zu Empfänger wählen. Da die Kanten des spannenden Baums des Eventmanagers auf Basis der Routingtabellen so gewählt wurden, daß jeweils die schnellsten Links zugrundeliegen, werden die Events mit hoher Wahrscheinlichkeit auch die vorgesehenen Wege nehmen. Dies trifft trivialerweise auf UDP Nachrichten zu. Für TCP-Verbindungen gilt dies ebenfalls, da TCP auf IP aufsetzt und somit keinen Einfluß auf eine feste Routenwahl hat.

> *Natürlich kann über die Optionen eines IP Paketes für Source Routing die Route vorgegeben werden. Wir wollen diesen Punkt aber außer acht lassen, da Source Routing für normale Datennachrichten nicht üblich ist und darüberhinaus von vielen Routern ignoriert wird, um das "Verstopfen" bestimmter Links des Netzwerks zu verhindern.*

Bei Zusammenbruch eines Links, der vom Eventmanager verwendet wird, geschieht nun folgendes: Die aktuell gesendete(n) Nachricht(en) sind verloren. Bei UDP wird die Nachricht nach Ablauf eines Timeouts oder nach Eintreffen der ICMP-Fehlermeldung wiederholt. TCP erledigt das protokollintern. Unter Umständen wird die bestehende TCP-Verbindung aber unterbrochen. Dies wird von beiden Seiten bemerkt und führt zu einem Neuaufbauversuch. IP sucht dann einen anderen Weg durch das Netz vom Sender zum Empfänger. Das ist grundsätzlich eine sehr positive Eigenschaft. Allerdings kann es unter Umständen nun billiger sein, wenn der bisherige Ast gekappt wird und die abgeschnittenen Knoten an einer anderen Stelle in den spannenden Baum wieder angefügt werden.

Netzpartitionierung

In diesem Fall kann nicht mehr jeder Knoten, der Teilnehmer des Eventchannels ist, erreicht werden.

Die Knoten an der "Bruchstelle" warten bis der ausgefallene Knoten wieder aktiv ist. Der Eventmanager versucht regelmäßig die Verbindung wieder aufzubauen. In der Zwischenzeit eintreffende Events werden gespeichert und nach Reparatur des Baums weitergesendet. Das Wissen, welche Nachbarn ein Knoten hat, ist ohnehin für das Versenden der Events nötig. Dies ist ein sehr einfaches und effektives Verfahren, das aber nur zufriedenstellend funktioniert, wenn der ausgefallene Knoten sehr schnell wieder aktiv wird.

Ein Nachteil dieser Lösung ist, daß der Eventmanager unter Umständen sehr viele Events irgendwo speichern muß. Dies ist aus Gründen des Speicherplatzverbrauchs nicht immer akzeptabel. Auch kommerzielle Systeme kommen beim einfachen Zwischenspeichern von Events für die Fehlerbehandlung schnell an ihre Grenzen. TIBCO Rendezvous gibt keine Garantien, wann ein Fehler gemeldet wird oder Events verloren gehen [TIBCO96]. IONA OrbixTalk liefert einen Fehler, wenn ein konfigurierbares Zeitintervall (standardmäßig 60 Sekunden) abgelaufen ist oder der konfigurierte Zwischenspeicher erschöpft ist [Orbix96].

Eine Möglichkeit wäre noch, den Eventchannel in solchen Fällen in beiden Partitionen zu schließen und die Fehlerbehandlung der Anwendung zu überlassen (Fail-Stop-Semantik). Natürlich ist dies für die Anwendung nicht besonders hilfreich.

II.8.2. Ausfall eines Knotens

Dieser Fehlerfall läßt sich in zwei Kategorien aufteilen: Ausfall des lokalen Eventmanagers und Ausfall des ganzen Rechner. Betrachten wir zunächst den zweiten Fall.

Ausfall des ganzen Rechners

Dies bedeutet, daß der Fehler in der Rechnerhardware liegt. Sehr wahrscheinlich wird der Rechner aufgrund des Hardwareproblems nicht in kurzer Zeit wieder verfügbar sein. Damit ist das Agentensystem nicht mehr lauffähig und Consumer und Supplier auf diesem Rechner beendet. Ein geordnetes Wiederaufsetzen des ganzen Systems ist ohne zusätzliche Maßnahmen (Sicherungspunkte, stabiler Speicher, eventuell Transaktionsverarbeitung auch für das Agentensystem, siehe auch [Jalote94]) nicht mehr möglich. Die Absicherung dieses Falles wird hier nicht weiter betrachtet.

Ausfall des Eventdämons

In diesem Fall ist der Prozeß des Eventdämons, und damit die Verbindung des Rechners zum globalen Eventmanager, aus irgendeinem Grund beendet worden. Das Agentensystem ist aber weiterhin aktiv und funktioniert. Supplier und Consumer sind ebenfalls vorhanden und funktionsfähig.

Es ist nun Aufgabe des lokalen Eventmanagers des Agentensystems, das Fehlen des Eventdämonen zu bemerken und geeignete Maßnahmen zum Neuaufsetzen durchzuführen. Im Idealfall können dafür bereits Mechanismen verwendet werden, die beim Start des Agentensystem ohnehin die Verbindung zum Eventmanager veranlassen (siehe dazu Kapitel II.7.2).

Zunächst muß der Eventdämon neu gestartet werden. Wenn mehrere Agentensysteme auf dem Rechner aktiv sind, so können diese unter Umständen gleichzeitig das Fehlen des Eventdämon bemerken und jeweils versuchen den Dämon neu zu starten. Das ist aber nicht weiter schlimm, da der Dämon für seine Arbeit einen eindeutig spezifizierten TCP-Serverport beziehungsweise einen UDP-Port benötigt. Kann ein Dämon diesen Port nicht öffnen, da er schon belegt ist (von einem anderen Dämon oder gar von einem fremden Programm), so beendet er sich wieder. Ist ein fremdes Programm auf diesem Port aktiv, so kann keine Eventkommunikation stattfinden, ist es ein Dämon, der bereits von einer anderen Engine auf dem selben Rechner gestartet wurde, so ist alles in Ordnung.

Nach dem Starten führt der Dämon seine Initialisierungen durch und liest von stabilem Speicher seinen letzten konsistenten Zustand ein. Dann ist das System wieder einsatzfähig und eventuell auf Nachbarknoten zwischengespeicherte Events werden nun ausgeliefert.

In diesem Szenario genügt es, den Eventdämon mit Sicherungspunkten zu versehen.

> *Man könnte auch einen Reparaturagenten zum betroffenen Agentensystem schicken, der die Rechte eines Systemagenten hat und den Fehler analysieren und beheben kann, soweit dies überhaupt möglich ist.*

II.8.3. Re-Konfiguration des spannenden Baums

Ein Knoten bemerkt den Ausfall eines Nachbarknotens, wenn das Senden eines Events an den Nachbarn nicht funktioniert hat. Wir untersuchen in diesem Abschnitt, wie durch geeignete Umkonfiguration des spannenden Baums darauf reagiert werden kann. Die Verfahren setzen voraus, daß die beteiligten Knoten Wissen über die lokale Topologie des Baums haben.

Umgehen

Da es unter Umständen einige Zeit dauern kann, bis das Agentensystem den ausgefallenen Eventdämon "bemerkt", kann den Nachbarn nicht zugemutet werden, alle Events aufzubewahren. Stattdessen wird eine "Abkürzung" eingerichtet. Jeder Knoten kennt dazu alle Knoten, die direkt "hinter" jedem Nachbarknoten liegen.

Bei Ausfall eines Knotens geschieht nun folgendes (siehe dazu Abbildung II.5):

- Es existiert ein spannender Baum, der Knoten A verwendet (1).
- Die Nachbarknoten B - E bemerken den Ausfall von Knoten A (2).

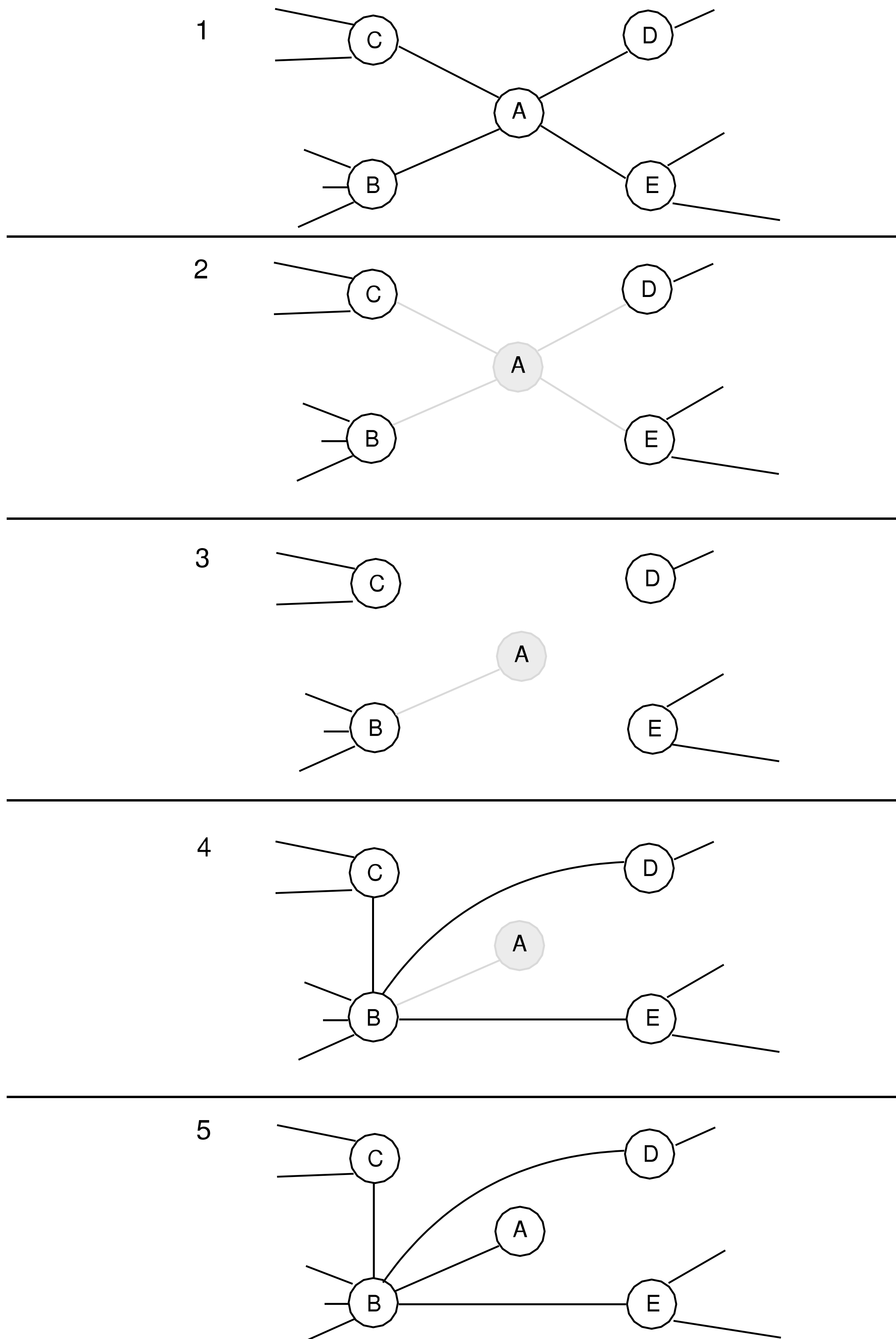


Abb. II.5: Lokale Umkonfiguration eines Eventbaums bei Ausfall von Knoten A unter Verwendung einer Umleitung. Ausgefallene Knoten und Kanten sind schattiert dargestellt.

- Der Knoten mit der kleinsten ID initiiert die Abkürzung. Wir nennen diesen Knoten den kleinsten Knoten. Da jeder beteiligte Knoten jeden Knoten kennt, der direkt über A erreicht wird, kann lokal in jedem beteiligten Knoten entschieden werden, welches der kleinste Knoten ist. Im Beispiel wird Knoten B die Abkürzung initiieren.
- Die Verbindung zum ausgefallenen Knoten wird, bis auf den kleinsten, bei allen Nachbarn gelöscht. Damit ist der spannende Baum nicht mehr zusammenhängend. Der ausgefallene Knoten ist nun logisch ein Blatt des spannenden Baums, der den kleinsten Knoten enthält (3).
- Der kleinste Knoten baut nun Verbindungen zu den anderen Nachbarknoten auf. Der spannende Baum ist nun zwar sehr wahrscheinlich nicht mehr optimal, aber zumindest wieder zusammenhängend. Sämtliche Events, die auch an den ausgefallenen Knoten gesendet werden müssen, werden beim kleinsten Knoten gespeichert (4).
- Sowie der ausgefallene Knoten wieder verfügbar ist, wird die Verbindung zum kleinsten Knoten aufgebaut. Auch A kann lokal entscheiden, welches der kleinste Knoten in seiner Nachbarschaft ist (5).

- Nach dem Wiederanlauf kann wieder der alte (meist bessere) Baum verwendet werden. Es muß dann aber sicher verhindert werden, daß kurzfristig Zyklen im Baum entstehen. Dies ist aber auf Basis einer eindeutigen Event-ID möglich.

Bestimmung der Qualität des "schlechteren" Baums

Andererseits kann man sich auch mit dem geringfügig schlechteren Baum zufriedengeben und diesen bis zur nächsten regulären Neukonfiguration des gesamten Baums beibehalten. Dazu muß man allerdings wissen, um wieviel schlechter der neue Baum gegenüber dem alten ist. Zur Berechnung der Differenz genügt es, nur die Kanten zwischen den betroffenen Knoten zu betrachten, da sich der restliche Baum nicht verändert. Das Problem ist im Grunde lediglich der lokale Vergleich der Gewichte zweier spannender Bäume zwischen den Nachbarknoten des ausgefallenen Knotens. Sei K_{vorher} die Konfiguration des Baums vor dem Ausfall (entspricht Bild 1) und $K_{nachher}$ die Konfiguration des Baums nach Ausfall und Neustart des ausgefallenen Knotens (entspricht Bild 5). Die Differenz ε der Summe der Kantengewichte von K_{vorher} beziehungsweise $K_{nachher}$ können wir nun als Maß für die Verschlechterung des Baums verwenden.

$$\varepsilon = \sum_{\text{Kantengewichte}} K_{nachher} - \sum_{\text{Kantengewichte}} K_{vorher} \qquad (II.17)$$

Die Berechnung der neuen Kantengewichte ist nicht übermäßig teuer, da nur die Kanten zwischen den betroffenen Knoten berücksichtigt werden müssen. Überschreitet der mit Formel *(II.17)* berechnete Wert eine vorher festgelegte Grenze, wird wieder auf den alten Baum umgeschaltet (mit dem dazugehörigen Aufwand), ansonsten bleibt bis zur nächsten Neukonfiguration der etwas schlechtere Baum erhalten.

Bewertung und Verfeinerung

Wenn der ausgefallene Knoten wieder verfügbar ist, kann sichergestellt werden, daß er alle Events zugestellt bekommt, da sie beim kleinsten Knoten gespeichert wurden. Events, die auf dem ausgefallenen Knoten liegen, können nach dem Wiederanlaufen weitergeschickt werden, da die Verbindung zum spannenden Baum (zumindest logisch) nie ganz verloren geht. Der einzige Nachteil des Verfahrens ist die Verzögerung bei Events, die in einer bestimmten Reihenfolge ausgeliefert werden müssen, wenn einer dieser Events direkt vor dem Abbruch auf dem ausfallenden Knoten ankommt und nicht mehr weitergesendet wird.

Der aufmerksame Leser wird sich gefragt haben, warum wir nicht die "billigste" Kante zum Wiedereinfügen des ausgefallenen Knoten verwenden. Der Grund dafür ist, daß sich das Gewicht der Kanten über die Zeit verändern kann. Ein ehemals sehr schneller Link kann beispielsweise durch steigenden Datenverkehr anderer Programme nun übermäßig belastet werden, so daß eine andere Verbindung durch das Netz zu einer Partition inzwischen billiger ist. Da wir einen eindeutigen Knoten bestimmen wollen, an den sich der ausgefallene Knoten nach Wiederanlauf wenden kann, müßte (um eine eindeutige Zuordnung zu erreichen) für den Ansatz der "billigsten Kante" der Zustand zum Zeitpunkt des Ausfalls als Kriterium herhalten. Wie gerade beschrieben besteht aber keine Garantie, daß diese Beschreibung zum immer noch zutrifft. Wir halten daher den beschriebenen Ansatz des kleinsten Knotens für leichter nachvollziehbar und auch einfacher zu implementieren, da er nur von statischen Gegebenheiten des Netzes Gebrauch macht.

Es fehlt noch die Behandlung des Falls, daß der kleinste Knoten ausfällt. Der vorgestellte Algorithmus geht davon aus, daß dies nicht eintrifft. Prinzipiell kann dies nur durch repliziertes Speichern der Events behandelt werden. Dies läßt sich zum Beispiel analog dem in [Kaashoek91] erwähnten Verfahren erledigen. Der Anwender eines Channels muß angeben wieviel Knoten gleichzeitig ausfallen dürfen, ohne daß die Funktionalität leidet. Sei diese Zahl n. Wir lassen dann anstatt nur des kleinsten Knotens einfach die $n+1$ kleinsten Knoten um einen ausgefallenen Knoten die Events speichern. Sollte ein Knoten weniger als $n+1$ Nachbarknoten haben, so werden die Events in einem größeren Umkreis um den ausgefallenen Knoten im Baum gespeichert. Die Nachbarknoten reichen an ihre eigenen Nachbarn die Information weiter, für wieviel Redundanz diese zu sorgen haben. Ist das nicht möglich (zum Beispiel weil dadurch ein Blattknoten für dreifache Redundanz sorgen müßte), so müssen die Nachbarn des ausgefallenen Knotens kooperativ die Informationen auf andere Knoten im Baum verteilen. Ist dies nicht möglich, so ist die Grenze der Fehlertoleranz erreicht. Der Vorteil dieses Verfahrens ist, daß es vollkommen ausreicht, wenn ein Knoten jeweils die Nachbarn seiner Nachbarn kennt. Das heißt alle Knoten, die er über den Baum mit 2 Sprüngen erreichen kann.

Neukonfiguration

Bei einer kompletten Neukonfiguration wird ein neuer Baum ohne den ausgefallenen Knoten erzeugt. Eine Neukonfiguration lohnt sich dann, wenn die gestiegenen Kommunikationskosten über den nun nicht mehr optimalen Baum in einem Zeitraum t_A teurer sind, als die Kosten für den Aufbau des neuen Baums. Abhängig von der Nutzung

des Baums, der Verschlechterung des alten Baums gegenüber einem optimalen Baum und t_A kann sich das schon nach kurzer Zeit lohnen. Algorithmisch ist eine Bestimmung des Break-Even-Points nur durch die tatsächliche Berechnung des neuen Baums und Abschätzung der Gruppenaktivität möglich. Dann ist der neue Baum aber auch schon berechnet und kann verwendet werden. Wenn es absehbar ist, daß der ausgefallene Knoten wahrscheinlich aufgrund von Hardwaredefekten ausgefallen ist (zum Beispiel wenn er über eine längere Zeit nicht mehr auf PING-Signale reagiert), sollte eine Neukonfiguration durchgeführt werden, um möglichst schnell wieder einen möglichst billigen Baum zu erhalten.

Prinzipiell genügt es, die entstandenen Partitionen wieder über billige Kanten zu verbinden. Dazu ist es nötig festzustellen, welche Kanten zwischen den Partitionen existieren und dann die jeweils billigsten auszuwählen. Dies entspricht dem letzten Schritt des Algorithmus von Gallager, Humblet und Spira [Gallager83], der auch als Grundlage zum Aufbau der Multicastbäume verwendet werden kann (zum Beispiel in Kapitel III.2.2 [Belkeir89]).

Umschalten auf den neuen Baum

Bleibt nur noch die Frage zu klären, wie auf den neuen Baum umgeschaltet wird, ohne daß Events verloren gehen oder verdoppelt werden (die Kommunikation innerhalb eines Baums ist ja bereits zuverlässig). Wall schlägt dazu eine einfache und doch effektive Methode vor [Wall80, S. 96f]. Er verwendet das Konzept des im ersten Teil beschriebenen Convergecasts. Wir benötigen zunächst einen Initiatorknoten, der das Umschalten startet. Dieser sendet auf dem alten Baum den Systemevent "Umschalten". Die Blätter des Baums starten daraufhin einen Covergecast "Annahmestop". Knoten, die diesen Convergecast erhalten, senden nun keine **neuen** Events mehr in den Baum, sondern speichern diese lokal in einer Warteschlange. Events, die sich bereits im Baum befinden, werden trotzdem weitergeschickt. Der Zentralknoten des Convergecasts sendet nun den Systemevent "LetzterEvent" in den Baum. Wenn dieser Event an den Blättern des Baums eintrifft, ist garantiert, daß keine alten Events mehr im Baum geroutet werden. Die Blätter initiieren nun den letzten Convergecast "Abbauen" auf dem alten Baum. Nach dessen Bearbeitung kann ein Knoten die Datenstrukturen des alten Baums vernichten. Der Convergecast konvergiert bei einem neuen Zentralknoten, der nun auf dem neuen Baum den Event "Wiederaufnahme" versendet, woraufhin der neue Baum mit den bereits wartenden Events gefüllt wird.

Beim Umschalten auf den neuen Baum entstehen somit folgende Kosten:

$$Kosten_{Umschalten} = 2 \times Kosten_{Event,\ alter\ Baum} + 2 \times Kosten_{Convergecast} + Kosten_{Event,\ neuer\ Baum} \qquad (II.18)$$

mit $Kosten_{Convergecast} = Kosten_{Event}$ ergibt sich

$$Kosten_{Umschalten} = 4 \times Kosten_{Event,\ alter\ Baum} + Kosten_{Event,\ neuer\ Baum} \qquad (II.19)$$

Zusätzlich müssen noch die Kommunikationskosten für die (verteilte) Berechnung des neuen Baums berücksichtigt werden.

Eine andere Methode zur Re-Konfiguration

Eine interessante, aber in dieser Arbeit nicht weiter untersuchte, Methode soll hier kurz skizziert werden [Baumann97b].

Bei dem Annahmestop-Convergecast werden die Nachbarschaftsbeziehungen aller Knoten eingesammelt. Der Zentralknoten kennt somit die Konfiguration des Gesamtsystems und kann damit lokal einen neuen spannenden Baum über diese Knoten berechnen. Die Informationen über den neuen Baum werden dann mit dem letzten Event an alle Knoten verteilt, die dann auf den neuen Baum umschalten. Als Sonderfall muß allerdings die Neueinbeziehung von Knoten, die nicht Mitglied im alten Baum sind, beachtet werden. Wir benötigen dann nur noch $2 \times N$ Nachrichten für die Berechnung des neuen spannenden Baums, da die verteilte Berechnung des spannenden Baums durch die lokale Berechnung im Zentralknoten ersetzt wird.

Dieser Ansatz hat den großen Vorteil, daß er in dicht vermaschten Graphen deutlich besser skaliert, als eine verteilte Berechnung mit dem Algorithmus von Gallager, Humblet und Spira [Gallager83].

II.9. Anforderungen auf einen Blick

Wir fassen die in diesem Teil erarbeiteten Anforderungen an einen Eventmanager nun kurz zusammen.

- **Verteilung**
  Aus Gründen der Ausfallsicherheit und Fehlertoleranz wollen wir auf zentrale Komponenten soweit wie möglich verzichten. Neue Knoten sollen jederzeit in das Eventmanagementsystem eingebunden werden können.

- **einfache Verwendbarkeit**
  Der Eventmanager soll dem Programmierer einer Anwendung die Arbeit vereinfachen und ihn soweit wie möglich unterstützen (zum Beispiel durch einfache Konfiguration des Systems).

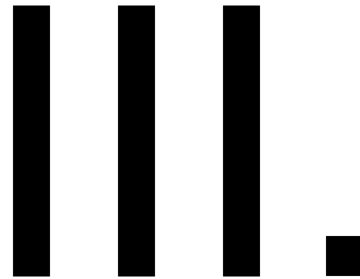
- **Eventchannels als Kommunikationsmedium**
  Events werden über Eventchannels ausgetauscht. Die Nutzer eines Eventchannels sollen eine gemeinsame Semantik besitzen. Pro Eventchannel kann ein eigenes Eventprotokoll verwendet werden, soweit das Protokoll dem Eventmanager bekannt ist.

- **Mobilität und Effizienz**
  Die Verteilung der Events soll, unter Beachtung der Tatsache, daß Consumer und Supplier jederzeit den Rechner wechseln dürfen und dabei keine Events verloren gehen sollen, möglichst effizient erfolgen.

- **Zuverlässigkeit**
  Für einige Anwendungen ist es wichtig, daß jeder Teilnehmer eines Eventchannels jeden darauf gesendeten Event erhält, egal wo sich der Teilnehmer befindet.

- **Reihenfolge**
  Es muß möglich sein, Events innerhalb eines Eventchannels zumindest kausal ordnen zu können. Grundsätzlich können Events vom selben Supplier einander nicht überholen.

- **Authentifikation und Autorisierung**
  Events müssen eine eindeutige, nachträglich nicht veränderbare Absenderkennung tragen.



III. Eventmanager

In diesem Teil der Arbeit werden existierende Produkte zum Eventmanagement kurz vorgestellt. Darüberhinaus werden bekannte Algorithmen und eigene Ideen untersucht. Abschließend vergleichen wir die Ansätze mit den Anforderungen aus dem vorangegangenen Teil und entscheiden, welcher Ansatz für eine Implementierung in einem System mobiler Software-Agenten am besten geeignet ist.

III.1. Der CORBA Event Service

Die Common Object Request Broker Architecture (CORBA) der Object Management Group (OMG) beschreibt, wie ein Client unter Verwendung eines Object Request Brokers (ORB), eine Operation bei einem eventuell entfernten Objekt anstoßen kann. Der ORB ist dabei für das Auffinden des passenden Objekts, für das Übertragen des Auftrags, sowie das Übertragen der Antwort zuständig. Darüberhinaus ist auch die Fehlerbehandlung Aufgabe des ORBs. Unter Verwendung von ORBs ist es verhältnismäßig einfach, verteilte Applikationen zu erstellen und auszuführen. Der Event Service ist eine Erweiterung des CORBA-Standards [OMG97].

Objekte, die Events versenden, werden "Supplier" und solche, die Events empfangen, "Consumer" genannt. Supplier und Consumer kommunizieren normalerweise indirekt über einen Eventchannel. Die OMG definiert nicht wie der Eventchannel aufgebaut ist. Er wird lediglich durch ein CORBA-Objekt repräsentiert. Es ist nicht festgelegt, ob der Eventchannel ein ganz normales Objekt ist, das irgendwo im Netz auf einem Rechner liegt, oder ob er verteilt realisiert ist. Eventchannels können typisiert oder untypisiert sein. Der Eventchannel verhält sich gegenüber einem Supplier wie ein Consumer und gegenüber einem Consumer wie ein Supplier. Jeder Consumer muß sich bei dem Supplier registrieren lassen, von dem er Events erhalten möchte.

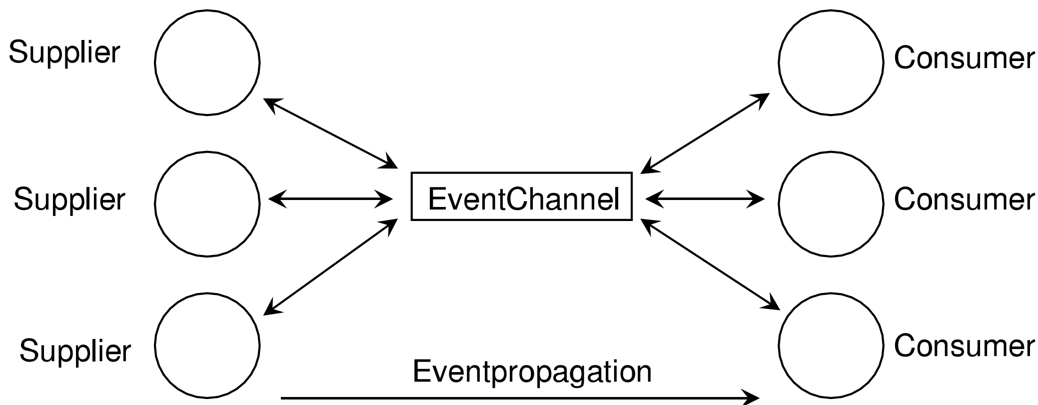


Abb. III.1: Der CORBA-Eventchannel mit Consumern und Suppliern

Die Kommunikation zwischen Consumer und Supplier kann auf zwei Arten erfolgen. Beim ***Pull***-Modell holt der Consumer beim Supplier Events ab. Beim ***Push***-Modell "schiebt" der Supplier Events zum Consumer. Die Kommunikation zwischen Supplier und Consumer ist immer synchron. Wie die Kommunikation innerhalb des Eventchannels abläuft ist nicht spezifiziert.

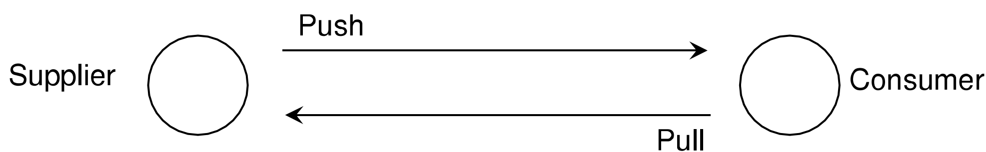


Abb. III.2: Push- und Pull-Modell bei CORBA

Folgende Eigenschaften machen das Event-Service-Modell der OMG mächtig:

1. Ein Eventchannel verhält sich je nach Kommunikationspartner wie ein Supplier oder Consumer und entkoppelt damit die kommunizierenden Objekte voneinander.

2. Dadurch können auch mehrere Eventchannels kaskadiert und somit ganze Hierarchien von Channels mit einem weiten Funktionsspektrum erzeugt werden.

3. Die Komplexität von Mulitcastkommunikation zwischen vielen Suppliern und Consumern ist vollkommen im Eventchannel gekapselt und einfach zu verwenden.

4. Der Eventchannel erlaubt es, die Kommunikationsmodelle miteinander zu mischen. Es ist also möglich, am selben Channel sowohl einen Push-Supplier als auch einen Pull-Consumer zu betreiben. Events die zwischen zwei Pulls des Consumers eintreffen werden vom Eventchannel gepuffert.

Die CORBA Event Services beschreiben ein mächtiges Modell zur Eventkommunikation. Man beachte, daß CORBA Supplier, Consumer und Eventchannel zum Teil sehr viel mehr können als in dieser Diplomarbeit vorausgesetzt.

Leider ist der Standard in einigen Bereichen sehr allgemein gehalten und bietet Interpretationsspielräume, die von kommerziellen Herstellern der ORBs unterschiedlich gelöst wurden. Beispielsweise beschreibt der Standard nicht, was mit Events passieren soll, wenn der Supplier schneller sendet, als der oder die Consumer die Events verarbeiten. Soll gepuffert werden oder nicht? Eventchannels von IONA OrbixTalk beispielsweise puffern solche Events [Orbix96]. Es gibt auch noch Schwierigkeiten, wenn man zwischen CORBA-konformen Systemen unterschiedlicher Hersteller Events austauschen möchte. Neuerdings gibt es eine Protokollspezifikation speziell für die Zusammenarbeit unterschiedlicher ORBs (Internet Inter ORB Protocol, IIOP).

Implementierungen der CORBA Event Services gibt es inzwischen von einer ganzen Reihe von Herstellern (IONA OrbixTalk, TIBCO TIB, IBM Blueprint, ...)

III.2. Basic Building Blocks

In diesem Kapitel werden existierende Ansätze und Verfahren für die Gruppenkommunikation vorgestellt, die zum Teil auch von den kommerziellen Lösungen als Grundlage verwendet werden. Wir können bei Betrachtung der Algorithmen Events mit Multicastnachrichten gleichsetzen.

III.2.1. IP Multicast und statisches Multicastrouting

Dieser Abschnitt beschreibt wie Multicast im Internet implementiert ist. Es gibt einen "Proposed Standard", der spezifiziert, wie Hosts an Multicastgruppen teilnehmen können und die nötigen Änderungen an der Implementierung des Internet Protocols beschreibt [Deering89]. Die meisten neueren IPv4-Implementierungen unterstützen diesen Standard.

Multicasting im Internet

IP-Multicast ist auf der Netzwerkebene angesiedelt. Eine Hostgruppe wird über eine gemeinsame IP-Adresse angesprochen. Ein IP-Datagramm wird an jeden Host in dieser Gruppe ausgeliefert, soweit dies möglich ist. IP übernimmt keine Garantie, daß das Datagramm bei jedem Host intakt ankommt. Ein Host kann einer Hostgruppe jederzeit beitreten oder sie verlassen. Es gibt zwei Arten von Hostgruppen:

- **permanent**
  Die IP-Adresse der Gruppe ist administrativ festgelegt. Eine permanente Gruppe kann null oder mehr Mitglieder haben.

- **transient**
  Alle IP-Multicast-Adressen, die nicht permanent sind, können von Hosts als Gruppenadresse frei verwendet werden. Diese Gruppen existieren nur solange, wie sie auch Mitglieder haben.

Multicastgruppen verwenden Klasse-D-IP-Adressen, bei denen die obersten 4 Bit auf 1 gesetzt sind. Sie umfassen den Adreßraum von 224.0.0.0 bis 239.255.255.255. 224.0.0.0 ist garantiert keiner Gruppe zugeordnet. 224.0.0.1 ist die permanente Gruppe aller IP-Hosts (einschließlich Gateways) im lokalen Netz. IP-Pakete mit dieser Zieladresse werden nur lokal ausgeliefert und nicht auf andere Netze weitergeleitet.

In IPv6 ist die Multicastaddressierung sauberer in das Addressierungsschema integriert. Alle Knoten werden über die Adresse FF0?::1 addressiert, alle Router über FF0?::2. Das '?' wird jeweils durch einen Wert für den Scope des Multicasts ersetzt: 1= lokal, 2=gleiches Netz, 5=gleiche Site, 8=gleiche Organisation, E=weltweit. Broadcastadressen gibt es in IPv6 nicht mehr.

Jeder Host wird, gemäß seiner Multicastfähigkeiten, in eine von drei Klassen ("Level of Conformance") eingeteilt.

- **Level 0**
  Der Host unterstützt kein Multicastrouting.

- **Level 1**
  Die IP-Schicht des Hosts gestattet das Senden von UDP-Paketen an Hostgruppen. Der Empfang von Multicastpaketen ist nicht möglich.

- **Level 2**
  Hosts in dieser Klasse können sowohl Multicasts senden, als auch empfangen. Dazu müssen sie IGMP implementieren, um Hostgruppen beizutreten und sie zu verlassen.

Jeder Host legt selber die Hostgruppen fest, zu denen er gehört. Das Senden einer Nachricht an eine Hostgruppe erfolgt analog dem normalen Senden einer Nachricht. Als Zieladresse wird die IP-Adresse der Hostgruppe angegeben. Die Nachricht wird im LAN mit dem normalen Unicast-Verfahren versendet. Ein spezieller Multicastrouter empfängt die Nachricht und leitet sie an andere Netze weiter, wenn die IP-Time-To-Live größer eins ist. Der Empfang einer Multicastnachricht läuft ähnlich. Jeder Host ist Mitglied der entsprechenden Multicastgruppe im LAN und empfängt so an die Gruppe gerichtete Nachrichten. Auch hier sorgt der Multicastrouter dafür, daß Nachrichten aus fremden Netzen lokal an die Gruppe ausgeliefert werden.

Von einer Anwendung können nur einzelne UDP-Nachrichten an die Gruppe gesendet werden. Eine sichere und zuverlässige Kommunikation im Stile von TCP ist im Standard nicht vorgesehen und muß durch die Anwendung implementiert werden (So macht es zum Beispiel der Event-Service von IONAs Orbix [Orbix96]).

Das Internet Group Multicast Protocol (IGMP)

Bisher haben wir noch nicht beschrieben, wie der Multicastrouter erfährt, welche Multicastgruppen in seinem LAN vorhanden sind. Zu diesem Zweck wird IGMP verwendet, das ein integraler Bestandteil der IP-Implementierung von Level-2-Hosts ist.

IGMP beschreibt zwei verschieden Arten von Nachrichten: Queries werden vom Multicastrouter regelmäßig als lokaler Broadcast in die Gruppe aller Hosts gesendet (224.0.0.1). Empfängt ein Host eine Query-Nachricht, so sendet er Reports aller Gruppen, denen er angehört. Report- und Query-Nachrichten sind sehr kurz. Sie enthalten pro Nachricht nur die IP-Adresse einer Gruppe. Wenn ein Host mehreren Gruppen angehört, sendet er mehrere Reports. IGMP beschreibt Maßnahmen, um zu verhindern, daß auf jeden Query Report-Stürme von den Hosts erzeugt werden (Näheres dazu in Appendix I von [Deering89]). Im Schnitt wird pro Query-Intervall und Gruppe ein Report im LAN gesendet.

Erhält ein Multicastrouter auf mehrere aufeinanderfolgene Queries zu einer ihm bekannten Gruppe keine Reports mehr, geht er davon aus, daß kein Host dieser Gruppe mehr im dem LAN vorhanden ist und schickt auch keine weiteren Multicastnachichten dieser Gruppe in dieses LAN. Der Multicastrouter braucht nicht zu wissen, welche Hosts in einem LAN

Mitglied einer Gruppe sind. Da er die Nachrichten mit lokalem Multicast in das LAN sendet, werden diese von allen interessierten Hosts empfangen und der Multicastrouter muß nur die Information verwalten, ob Mitglieder einer Hostgruppe in dem LAN sind oder nicht.

Jenseits des Multicastrouters

Der Multicastrouter kennt unter Verwendung von IGMP alle Gruppen in seinem LAN. Wir beschreiben nun kurz das im Internet verwendete Verfahren zum Weiterleiten der Multicastpakete in alle LANs, in denen Hosts der entsprechenden Multicastgruppe aktiv sind. Der verwendete Algorithmus ist ausführlich in [Deering88] beschrieben. Er baut auf den Distance-Vector-Informationen und den Routingtabellen des Unicastroutings auf. Router verwalten in ihrer Routingtabelle für jedes bekannte Ziel eines Pakets einen Eintrag mit Informationen über die Entfernung zum Ziel ("distance", gemessen in Hops) und dem nächsten Rechner auf dem kürzesten Weg zum Ziel. Jeder Router sendet periodisch auf allen angeschlossenen Netzwerken Nachrichten der Form (Ziel, Entfernung) für jedes bekannte Ziel. Aufbauend auf diesen "Distanz-Vektoren" aktualisieren alle Router ihre Routingtabelle.

Im Gegensatz zu den in dieser Arbeit beschriebenen Verfahren, werden die Informationen der Routingtabelle nicht direkt zum Aufbau eines spannenden Baums über alle Knoten der Hostgruppe verwendet. Stattdessen wird für jeden möglichen Sender implizit ein eigener Baum verwaltet, dessen Wurzel der Sender der Multicastnachricht ist. Ein Router verbreitet eine Multicastnachricht nur dann weiter, wenn sie auf dem Link eingeht, der auf dem kürzesten Pfad zum Sender der Nachricht liegt. Genau diese Information wird in den Unicast-Routing-Tabellen gespeichert, um eine Nachricht möglichst schnell ans Ziel zu bringen. Die Einträge der Tabellen werden "in der Gegenrichtung" verwendet, deshalb spricht man hier auch von "Reverse Path"-Algorithmen. Im Basisalgorithmus ("Reverse Path Flooding") wird eine Multicastnachricht auf allen Links des Routers weitergesendet, außer auf jenem, über den sie empfangen wurde. Dies entspricht in Realität einem Broadcast der Nachricht an alle Hosts und führt in Netzwerken mit hoher Konnektivität leicht zu Duplikaten. Darüberhinaus wird unnötig viel Netzwerkbandbreite verschwendet, da Nachrichten auch in Subnetze weitergeleitet werden, in denen gar keine Teilnehmer der Hostgruppe vorhanden sind.

Beim "Truncated Reverse Path Broadcast"-Algorithmus (TRPB) werden daher alle Netze abgeschnitten, die ein Blatt des Multicastbaums darstellen. Ein Netz ist dann ein Blatt des Baums, wenn es keine Hosts der Multicastgruppe und keine weiteren Router, an die die Nachricht weitergeleitet werden muß, enthält. Jeder Router sendet periodisch auf jedem seiner angeschlossenen Links eine Nachricht der Form "Dieser Link ist mein nächster Hop zu diesen Zielen...". Somit kann jeder andere Router feststellen, ob er auf dem kürzesten Weg zu einem dieser Ziele liegt. Darüberhinaus weiß er über IGMP, ob im Subnetz Hosts der Gruppe vorhanden sind und kann so feststellen, ob in das betrachtete Subnetz Multicastnachrichten von dem betrachteten Ziel weitergeleitet werden müssen und ob das Subnetz ein Blatt des Multicastbaums ist. Dieser Algorithmus ist verhältnismäßig einfach

zu implementieren und ist ausreichend, wenn Durchsatz und Bandbreite der Blattnetzwerke die kritischen Ressourcen sind.

Tatsächlich verwendet wird im heutigen Internet der "Reverse Path Multicasting"-Algorithmus (RPM), der eine logische Verfeinerung von TRPB darstellt [Waitzman88]. Hier werden nun auch die nicht benötigten Äste des Multicastbaums zwischen den Routern abgeschnitten. Benötigt keines der Blatt-Subnetze eines Routers Multicastnachrichten einer Hostgruppe, so meldet der Router dies an den benachbarten Router weiter, der auf dem kürzesten Weg zum betrachteten Sender liegt ("Non Membership Report", NMR). Aufgrund der NMRs und IGMP kann jeder Router entscheiden, auf welchen seiner Ausgänge Multicastnachrichten eines bestimmten Senders weitergeleitet werden müssen und auf welchen nicht. Der Multicastbaum zu einem bestimmten Sender wird rekursiv solange beschnitten, bis nur noch die Äste übrig sind, die auch tatsächlich zu Empfängern führen. Deering und Waitzman beschreiben in der Literatur einige Erweiterungen und Feinheiten des Algorithmus, die für den Alltagsbetrieb notwendig sind.

Bewertung

IP Multicast ist ein genial einfaches Verfahren, zur Implementierung von Multicast im Internet. Es verwendet bereits im Netz vorhandene Informationen und Funktionen und beschränkt sich auf die minimal benötigen Erweiterungen, gemäß dem Motto "keep it simple". Der beste Beweis für die Verwendbarkeit des Systems sind regelmäßige Videoübertragungen von NASA, Videokonferenzen und Fernsehsendern rund um den Erdball unter Verwendung von IP Multicast.

Allerdings hat der verwendete Ansatz auch einige Nachteile.

- Die Festlegung der Hostgruppe ist mit dem Routing der Nachrichten an diese Gruppe eng verbunden. Es gibt keine Trennung von Übertragungsmechanismus und Übertragungspolitik.

- Es gibt keinerlei Zugangsbeschränkungen. Jeder kann sich in eine laufende Videokonferenz einklinken, solange er das richtige Programm verwendet (frei verfügbar) und weiß, welche Multicastadresse verwendet wird.

- IP Multicast kennt keine Fehlerkorrektur. Die Zuverlässigkeit eines Multicastpakets entspricht der eines UDP-Pakets.

- Zum derzeitigen Zeitpunkt sind lediglich etwa 30 % der Router des Internets multicastfähig. Zur Überbrückung der "Lücken" müssen handkonfigurierte Tunnels zwischen den beteiligten Netzen eingerichtet werden. Ohne einen solchen Tunnel ist beispielsweise lokale Multicastkommunikation zwischen Netzen der Fakultät Informatik an der Universität Stuttgart nicht möglich! Dieses Problem wird sich mit der Zeit und mit der Einführung von IPv6 entschärfen.

III.2.2. Dynamisches Multicastrouting

Wie im zweiten Teil dieser Arbeit bereits beschrieben, kann ein (minimal) spannender Baum zwischen den Knoten einer Gruppe die Übertragungskosten zum Versenden eines Events klein halten. Es wurde auch gezeigt, daß in einem System mobiler Agenten die Migrationsfrequenz eines Consumers sehr hoch sein kann. In diesem Abschnitt stellen wir einen inkrementellen Algorithmus vor, der unter den gegebenen Bedingungen die Kosten für die Aktualisierung des Baums klein hält. Der Algorithmus basiert auf einem Vorschlag von Ahamad und Belkeir [Belkeir89]. Er wurde um ein Primitiv für Migration erweitert und die Notation auf Eventchannels angepaßt. Im Gegensatz zu den im vorigen Abschnitt angeschnittenen Algorithmen gehen Ahamad und Belkeir davon aus, daß sich die Gruppenzugehörigkeit der Knoten regelmäßig ändern kann und dies auch schnell in der Kommunikationsstruktur berücksichtigt werden soll.

Der Algorithmus ist zweigeteilt. Für jedes (Sub-)Netz in dem mindestens ein Agentensystem aktiv ist, wird ein Koordinator (nicht zu verwechseln mit dem "Group Coordinator" der Agentengruppen!) bestimmt. Dies kann zum Beispiel ein Eventdämon eines Rechners des lokalen Netzes oder ein dedizierter Serverprozeß in einem Router sein. Die Anzahl der Koordinatoren ist also deutlich kleiner als die Anzahl der insgesamt beteiligten Rechner und verändert sich nur selten. Zwischen allen Koordinatoren existiert ein statischer spannender Baum, der beispielsweise mit dem Algorithmus von Wall [Wall80] oder dem von Gallager, Humblet und Spira [Gallager83] erzeugt werden kann. Da Koordinatoren nur selten gestartet und beendet werden, ist es akzeptabel für die Grundkonfiguration einen statischen Algorithmus zu verwenden. Die Koordinatoren können auch das in Kapitel II.5.2 beschriebene Kernsystem zum Speichern von persistenten Events realisieren. Der spannende Baum der Grundkonfiguration wird von Zeit zu Zeit neu berechnet. Ein Kriterium zur Neuberechnung geben wir in Kapitel III.2.4 .

Für jeden Channel, den ein Knoten kennt, unterhält dieser eine Datenstruktur namens ChannelRec, die eine Liste der Nachbarknoten bezüglich des Channels beinhaltet, sowie einen Merker, ob der Knoten Mitglied des Channels ist ("member") oder lediglich Events des Channels weiterleitet, also auf dem Weg zwischen zwei Channelteilnehmern liegt ("non-member").

Die Algorithmen sind in Anhang B formal beschrieben.

Restricted Broadcast

Der Restricted-Broadcast-Algorithmus (RB) wird zwischen den Koordinatoren ausgeführt. Ein Koordinator ist Teilnehmer eines Channels, wenn sich in seinem Verantwortungsbereich ein Channelteilnehmer befindet. Die einzelnen Multicastbäume der Channels verwenden nur den bestehenden spannenden Baum zwischen allen Koordinatoren (deswegen die Bezeichnung "restricted" Broadcast). Der Baum eines Channels ist somit ein Teilbaum des statischen Baums, bei dem alle Äste abgeschnitten sind, die nicht zu einem Channelteilnehmer führen. Bezüglich eines Channels kann ein Koordinator 3 Operationen ausgeführen. Die Bezeichnungen der versendeten Nachrichten sind jeweils kursiv gedruckt.

- **Channel beitreten**
  Dieser Fall tritt ein, wenn auf einem Host im Veranwortungsbereich des Coordinators ein Objekt dem Channel beitreten möchte und der dortige Eventdämon den Channel nicht kennt. Wenn der Koordinator den Channel auch noch nicht kennt, so legt er den ChannelRec an und sendet an seine Nachbarn über den statischen Baum einen *join-request*. Erhält er die Antwort *participant* von einem Nachbarn, so fügt er diesen in die Nachbarliste des ChannelRec ein. Empfängt ein Knoten *join-request* für einen Channel und kennt er diesen Channel noch nicht, so sendet er *join-request* an alle anderen Nachbarn im statischen Baum weiter. Erhält er von mindestens einem Nachbarn die Antwort *participant* oder kennt er den Channel bereits, so sendet er als Antwort *participant* an den Sender der Anforderung und trägt den entsprechenden Nachbarn in seine Nachbarliste ein. Ansonsten sendet er *non-participant*. Kennt ein Knoten den gesuchten Channel nicht, so wird dieser über den statischen Baum gesucht und bei Erfolg der Weg zum nächsten Knoten, der den Channel kennt, im Baum markiert.

- **Channel verlassen**
  Der Koordinator verläßt einen Channel, wenn der letzte Eventdämon in seinem Verantwortungsbereich den Channel verläßt. Wenn ein Koordinator mindestens zwei Nachbarn im Channel hat, so muß er weiterhin für das Weiterleiten von Events sorgen. Er ändert in seinem ChannelRec lediglich die Mitgliedschaft auf non-member. Wenn der Koordinator nur einen Nachbarn hat, so bildet er ein Blatt des Channel-Baums. Er sendet einen *delete-request* an diesen Nachbarn und löscht ChannelRec. Empfängt ein Koordinator *delete-request*, so löscht er den Sender aus der Liste der Channelnachbarn. Bildet er damit selber nun ein Blatt des Baums und ist nicht Channelmitglied, so sendet er *delete-request* an den einzigen verbliebenen Nachbarn weiter. Damit wird der Ast rekursiv bis zum nächsten Channelnachbarn gekürzt, der entweder selber Channelmitglied ist oder mehr als einen Nachbarn hat und damit für die Weiterleitung von Events zuständig ist.

- **Event senden**
  Wenn der Koordinator den Channel kennt, dann sendet er den Event an alle lokalen Mitglieder des Channels und an alle Nachbarn aus der Liste in ChannelRec. Kennt der Koordinator den Channel nicht, so sucht er einen Teilnehmer des Channels, ähnlich dem Verfahren bei Channelbeitritt. Empfängt ein Koordinator einen Event, so liefert er ihn lokal aus und sendet ihn an alle Nachbarn außer dem Koordinator, von dem er den Event erhalten hat. Events werden also entlang des beim Beitreten markierten Weges im Baum verteilt.

Optimierung

Der Algorithmus hat in der vorgestellten Form ein Performanceproblem bei kleinen Channels in einem weit verteilten System. Da der *join-request* erst bei einem Koordinator endet, der den Channel kennt und dies bei kleinen, geographisch eng begrenzten Channels nicht sehr viele sind, sind die Kosten und die Verzögerung für das Beitreten zu einem Channel sehr hoch. Die Kosten für das Finden des Channels sind ebenfalls hoch, wenn ein Knoten, der nicht Mitglied des Channels ist, in diesen einen Event senden möchte. Beide Probleme

werden durch eine leichte Modifikation des Algorithmus behoben. Jeder Knoten merkt sich für jeden Unterbaum, welche Channels dort vorhanden sind. Der *join-request*, beziehungsweise der "von außen" in den Channel gesendete Event, wird solange in Richtung Wurzel des Baums gesendet, bis ein Knoten den gesuchten Channel kennt. Dies ist trivialerweise für die Wurzel erfüllt. Dann geht es abwärts in den Unterbaum, bis ein Teilnehmer des Channels gefunden ist.

Der zweite Fall, das Senden eines Events in einen Channel von außerhalb, soll im tatsächlichen Anwendungsfall mit MOLE nicht vorkommen, da ein Supplier in einen Channel erst senden darf, wenn er dort auch Mitglied ist. Diese Bedingung ist für einige Dienstgüteparameter zwingend erforderlich.

Wo ist die Wurzel?

Bei IP Multicast ist der Sender einer Nachricht automatisch die Wurzel des zugehörigen Baums. Bei Restricted Broadcast wird stattdessen ein einzelner Baum für alle Kommunikationsbeziehungen verwendet. Ein Baum über einer Menge von Knoten hat die Eigenschaft, daß man einen beliebigen Knoten wählen und festhalten kann. Die restlichen Knoten hängen dann an den verbindenden Kanten "nach unten". Somit kann jeder beliebige Knoten die "Wurzel" des Baums sein. Die Wurzel des Baums sollte sinnvollerweise so angeordnet sein, daß der Baum möglichst ausgeglichen ist. Zum einen um die Verzögerung bis ein Event alle Knoten des Baums erreicht hat, möglichst gering zu halten, und zum anderen, um den benötigten Speicherplatz der Optimierung für das Finden von Gruppen klein zu halten. Wie bestimmen wir möglichst billig eine gute Wurzel des Baums?

Glücklicherweise hat der Algorithmus von Gallager, Humblet und Spira [Gallager83] die angenehme Eigenschaft eine ausgezeichnete Kante zu bestimmen, die im Herzen des Baums liegt. Gallager nennt diese Kante den Kern ("core") des Baums. Einer der zwei Knoten an dieser Kante übernimmt dann die Rolle der Wurzel des Baums. Soweit möglich ist der damit entstehende Baum ausgeglichen, womit das Problem gelöst ist.

Nearest Insertion

Im Originalpaper zu Restricted Broadcast [Belkeir89] wird für die Behandlung der Events im lokalen Netz ein weiterer Algorithmus vorgestellt ("Nearest Insertion"). Der "Nearest Insertion"-Algorithmus berechnet für ein lokales Netz bestehend aus Punkt-zu-Punkt-Verbindungen zwischen den Knoten den minimal spannenden Baum zwischen den beteiligten Knoten. Der Algorithmus ist aber für bus-basierte lokale Netze unnötig, da dort jeder Host direkt erreicht werden kann. Der Koordinator benötigt lediglich die Information, ob beziehungsweise wieviele Teilnehmer eines bestimmten Channels im lokalen Netz vorhanden sind.

Da die meisten lokalen Netze auf einem Bus basieren beziehungsweise diesen simulieren, vernachlässigen wir im Rahmen der Implementierung diesen Algorithmus und kommen direkt zur Behandlung der Mobilität von Eventconsumern. Der "Nearest Insertion"-Algorithmus ist der Vollständigkeit halber in Anhang B beschrieben.

Mobile Object Multicast Protocol (MOMP)

Dieses Protokoll wird auf Ebene der Eventdämonen ausgeführt. Seine Hauptaufgabe ist sicherzustellen, daß jeder Event einen Agenten an dessen derzeitigem Aufenthaltsort erreichen kann.

Prinzipiell arbeitet der Algorithmus analog zum Soft-Handover von Mobilfunknetzen. Migriert ein Agent zu einer anderen Location, so richtet der Eventdämon des Quellhosts eine Abkürzung für Events zum Eventdämonen des Zielhosts ein. Bereits während der Migration des Agenten wird eine Verbindung vom Zielhost zum Multicastbaum des Channels aufgebaut. Events, die während der Migration des Agenten und des Verbindungsaufbaus beim Quellhost für Channels des Agenten eintreffen, werden über die Abkürzung weitergeleitet.

Sowie der neue Ast des Multicastbaums aufgebaut ist, erhält der Zielhost auf zwei Wegen Events. Zum einen auf dem regulären Weg über den Baum und zum anderen über die Abkürzung vom Quellhost. Der Zielhost speichert zusammen mit der Adresse des Quellhosts die EventID des ersten Events, den er nach Aufbau einer Abkürzung in den Eventchannel sendet. Dieser Event muß irgendwann über die Abkürzung wieder beim Zielhost eintreffen, da über sie im Baum ein Zyklus entsteht. Sobald der Event eintrifft, kann die Abkürzung abgebaut werden.

Der Algorithmus

Im folgenden wird der MOMP-Algorithmus im Detail erläutert. Der Code ist wieder in Anhang B gegeben. In der Beschreibung gehen wir davon aus, daß die Eventdämonen den Coordinator direkt erreichen können (busbasiertes lokales Netz, zum Beispiel "Ethernet" nach IEEE 802.3) und der NI-Algorithmus daher nicht nötig ist. Falls ein lokales Netz keinen effizienten lokalen Multicast zur Verfügung stellt, kann der NI-Algorithmus aber problemlos als Zwischenschicht eingefügt werden.

Die Verwaltungsdatenstruktur für einen MOMP-Channel ist ähnlich aufgebaut wie beim Restricted Broadcast. Es wird eine Liste der Knoten verwaltet, zu denen gerade eine Umleitung durchgeführt werden muß.

- **Channel beitreten**
  Ist der gewünschte Channel beim Eventdämonen noch unbekannt, so sendet er *join-channel* an den Koordinator, um dort den Channelbeitritt des RB-Algorithmus auszulösen. Bei jeden Beitritt wird ein Zähler für die Anzahl der lokalen Mitglieder des Kanals erhöht. Die weitere Verwaltung der lokalen Mitglieder ist der besseren Übersicht wegen nicht dargestellt.

- **Channel verlassen**
  Fordert das letzte lokale Mitglied den Eventdämon zum Verlassen des Channels auf, so wird das dem Koordinator gemeldet und die Datenstrukturen des Channels freigegeben. Wenn noch Abkürzungen auf diesen Host verweisen, werden diese freigegeben.

- **Event senden**
  Ein Event des Channels wird direkt an alle Knoten in der Abkürzungsliste ("short cut list") gesendet, an die hostlokalen Mitglieder des Channels ausgeliefert und an den Koordinator zur Weiterverarbeitung gesendet. Die Sendung an den Koordinator kann auch mit lokalem Multicast erfolgen. Dann muß der Koordinator den Event nicht ein zweites Mal über den lokalen Bus an die anderen Eventdämonen im selben Netz senden. Events, die vom Koordinator im lokalen Netz verteilt werden müssen, können ebenfalls mit lokalem Multicast versendet werden.

- **Objektmigration behandeln**
  Dies ist der Kernpunkt von MOMP. Ein migrierendes Objekt kennt immer seinen Zielknoten, der dem Eventdämon zu Beginn der Migration mitgeteilt wird. Der Eventdämon sendet *shortcut-added* zum Zielknoten und trägt diesen in seine Abkürzungsliste ein. Sowie der Zielknoten *shortcut-added* empfängt, beginnt er mit dem Beitritt zum angegebenen Channel und speichert den Sender von *shortcut-added* in der Verweisliste ("remoteShortcutsFrom") zusammen mit der nächsten EventID des Channels auf diesem Host. Trifft ein Event mit dieser ID über die Abkürzung ein, so kann diese abgebaut werden und der Zielknoten sendet *clear-shortcut* zum Initiator der Abkürzung. Dieser entfernt den Zielknoten von seiner Abkürzungsliste. Wenn er keine lokalen Teilnehmer mehr hat, verläßt er selber den Channel und beendet alle noch auf ihn verweisenden Abkürzungen. Damit wird auch der Fall behandelt, daß eine Abkürzung zu einem Zielrechner eingerichtet wird und von dort weiter zu einem anderen Rechner, ohne daß auf den Zwischenstationen ein Event versendet wird, der für den Abbau der Abkürzung zwingend erforderlich ist.

Betrachtung der Fehlerfälle

Wir wollen nun zeigen, daß das in Kapitel II.6 beschriebene Loch bei der Auslieferung von Events an den Agenten bei Verwendung von MOMP nicht mehr auftreten kann. Voraussetzung dafür ist, daß normale Datenevents einander während der Verteilung durch den Eventmanager auf den einzelnen Hops und bei der Verarbeitung in den Knoten nicht überholen können. Für die Hops läßt sich das sicherstellen, indem für die Verbindungen zwischen den einzelnen Knoten des Baums ein Protokoll verwendet wird, daß die Ordnung der Events aufrechterhält (TCP oder UDP mit Sequenznummern pro Hop). Innerhalb der Knoten werden Events, die über denselben Link empfangen wurden, in der Reihenfolge des Empfangs behandelt und weiterverteilt.

Für die Betrachtung des Fehlerfalls nehmen wir folgendes Szenario an: Ein Agent migriert von Knoten A nach Knoten B, dabei wird eine Abkürzung eingerichtet. Ein Knoten C sendet einen Event E_1, der bei Knoten B eingeht und verarbeitet wird, bevor der Agent migriert. Um den Fehlerfall zu erzeugen, muß E_1 im Baum so verzögert werden, daß er erst nach Abbau der Abkürzung bei Knoten A empfangen und damit nicht an den Agenten ausgeliefert wird. Der Abbau der Abkürzung wird durch einen zweiten Event E_2 initiert, der entweder von Knoten A oder B, oder einem beliebigen anderen Knoten gesendet werden kann. Wenn E_2 sowohl über die Abkürzung, als auch über den Baum empfangen wurde, kann die Abkürzung abgebaut werden.

Mögliche Fehlerfälle

- **Knoten A sendet E_2**
  Der Quellknoten A kann E_2 erst senden, wenn die Abkürzung aufgebaut ist. E_2 wird über den Baum und die Abkürzung gesendet und von Knoten B empfangen. Daraufhin wird die Abkürzung abgebaut. In Abbildung III.3a zeigen wir, wie E_1 verzögert werden muß, damit der Agent E_1 nicht empfängt. Die verwendete Netztopologie ist angegeben. Die Position des Agenten ist durch einen fetten Balken markiert.

- **beliebiger Knoten sendet E_2**
  Ein Knoten D sendet E_2 zu einem beliebigen Zeitpunkt über den Baum. E_2 wird auch von A empfangen, welcher E_2 über die Abkürzung sendet und damit den Abbau der Abkürzung einleitet. In Abbildung III.3b zeigen wir, wie E_1 dann verzögert werden muß. Die verwendete Netztopologie ist etwas größer als im vorigen Abschnitt.

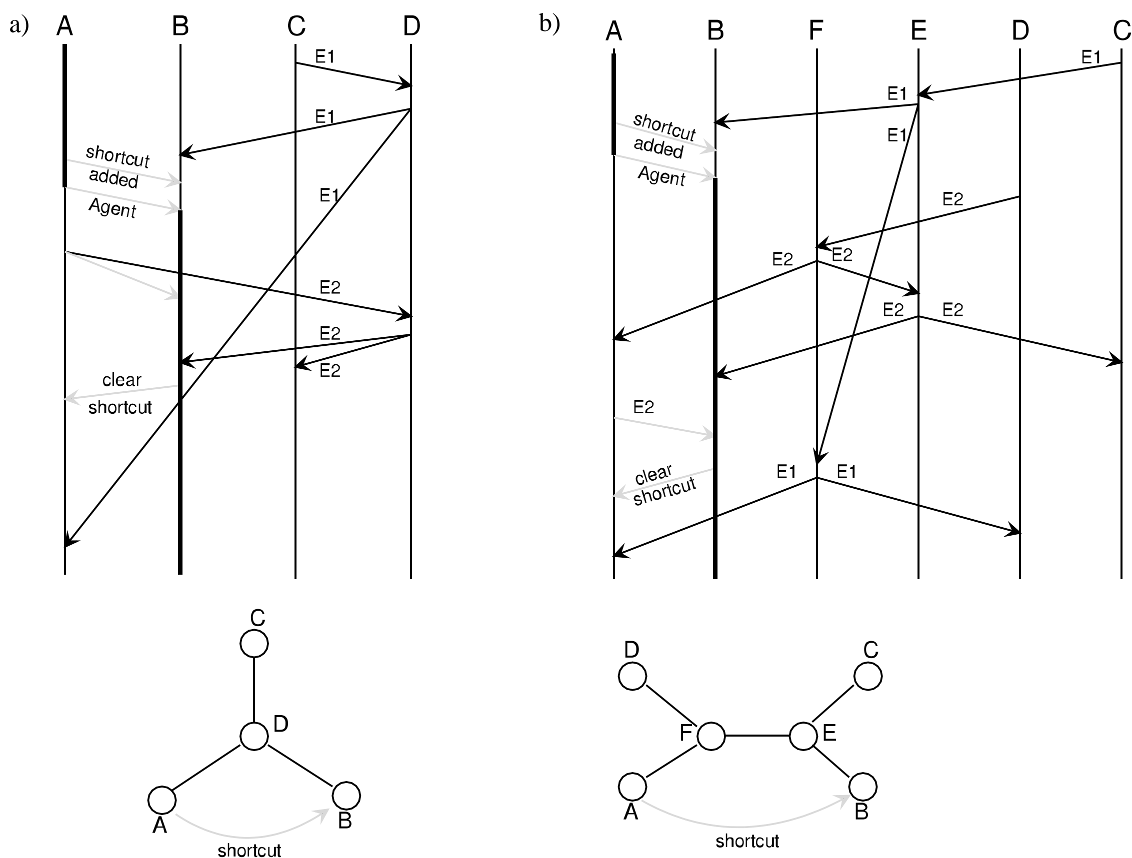


Abb. III.3: Die möglichen Fehlerfälle. Die verwendete Netztopologie ist jeweils angeben. In Beispiel a) wird der von Knoten D an Knoten A gesendete Event E_1 solange verzögert, bis die Abkürzung abgebaut ist. Analog wird in Beispiel b) der Event E_1 von Knoten E an Knoten F verzögert. Zu beachten ist, daß in keiner Richtung Events überholt werden. Der Aufenthaltsort des Agenten ist mit einem fetten Balken markiert. Über die Abkürzung gesendete Events sind grau gezeichnet.

Die Lösung mit MOMP

- **Knoten B sendet E_2**
  E_2 wird erst gesendet, wenn die Abkürzung bereits besteht. E_2 beschreibt einen Zyklus durch den Baum und über die Abkürzung zurück zu B. Alle Events, die vor E_2 in derselben Richtung im Baum übertragen werden, werden vor E_2 über die Abkürzung gesendet (wegen der Eigenschaft, daß Events einander nicht überholen können) und an den Agenten ausgeliefert. Events, die entgegen der Richtung von E_2 im Baum übertragen werden, kommen nach endlicher Zeit über den regulären Weg auf Knoten B an, da B Channelmitglied ist, und werden somit an den Agenten ausgeliefert.
  Der von B im Zyklus gesendete Event E_2 verhindert, daß durch extreme Verzögerungen Events nicht an den Agenten ausgeliefert werden können. Seine Position im Datenstrom des Baums teilt die Events in zwei Gruppen. Jene "vor" ihm werden noch über die Abkürzung an den Agenten gesendet, jene "nach" ihm über den regulären Baum.

Somit kann das Loch in der Eventauslieferung an einen migrierten Agenten nicht mehr auftreten.

Vorteile und Nachteile dynamischen Multicastroutings

\+ **beliebige Gruppenbezeichnungen**
Da der Algorithmus keine Vorgaben in Bezug auf Gruppenbezeichnungen macht sind prinzipiell beliebige Gruppenbezeichnungen möglich. Es sollte lediglich darauf geachtet werden, daß ein Event garantiert nicht mehr als ein IP-Paket benötigt.

\+ **reagiert schneller auf Änderungen in den Gruppenzuordnungen**
Durch die direkte Adaption der Multicastbäume, kann der vorgestellte Algorithmus sehr schnell auf Ortsveränderungen von Gruppenmitgliedern reagieren und trotzdem unter zeitweiser Vernachlässigung der Optimalitätskriterien die Zustellung der Events garantieren.

\+ **leichte Implementierbarkeit in Anwendungsschicht**
Im Rahmen dieser Diplomarbeit können keine Änderungen an der Netzwerksoftware von Routern oder Maschinen innerhalb oder außerhalb der Fakultät Informatik vorgenommen werden. Daher müssen die benötigten Algorithmen zu Demonstrationszwecken oberhalb der Transportschicht des Internets implementiert werden.

\- **Multicast bereits im Internet vorhanden**
Die hier vorgestellten Algorithmen können zwar mit dem IP-Multicast kombiniert eingesetzt werden, sind aber zu dem im Internet eingesetzten Multicastverfahren inkompatibel.

III.2.3. Kausale Ordnung von Events

Im zweiten Teil haben wir, vor allem aufgrund der Anforderungen aus Synchronisierungsanwendungen, festgestellt, daß Events innerhalb eines Eventchannels kausal geordnet sein sollen.

Im vorigen Abschnitt haben wir untersucht, wie Events zwischen den an einem Channel beteiligten Hosts verteilt werden. Birman und Joseph beschreiben in [Birman87] Theorie und Implementierung von Broadcastprimitiven. Der im folgenden vorgestellte Algorithmus basiert auf der Implementierung des dort eingeführten CBCAST-Primitivs. Der Algorithmus stellt die kausale Ordnung von abhängigen Events sicher, allerdings ohne globale Ordnung kausal unabhängiger Events.

Das folgende Beispiel zeigt, welche Auswirkung das für eine reale Anwendung hat: An einer Knotengruppe seien drei Rechner A, B und C beteiligt. A sendet eine Nachricht a, woraufhin B eine Nachricht b generiert. C sendet derweil eine davon unabhängige Nachricht c. Die Nachrichtenverarbeitung kann nun an den drei Rechnern in der jeweils angegebenen Reihenfolge erfolgen, ohne daß die kausale Ordnung zwischen den Nachrichten a und b zerstört wird:

Rechner A: a c b
Rechner B: a b c
Rechner C: c a b

Der Algorithmus

Die Grundidee des Algorithmus ist einfach. Zum Senden eines Events E wird ein Transportpaket konstruiert, das alle Events E_i enthält, die vor E ausgeliefert werden müssen. Daraus folgt, daß wenn E beim Empfänger ausgeliefert werden soll, alle E_i , die aufgrund der kausalen Ordnung vor E ausgeliefert werden müssen, spätestens zusammen mit E in die Auslieferungswarteschlange des Empfängers eingetragen werden.

Die Auslieferung von Events innerhalb eines Rechners erfolgt in unserem Modell immer synchron und in der gleichen Reihenfolge. Daher genügt es, wenn der Eventdämon die Reihenfolgesicherung der Events durchführt. Pro Eventdämon gibt es einen Eventpuffer BUF, der alle Events enthält, die von ihm verarbeitet werden. Das sind Events, die auf diesem Rechner erzeugt wurden, dort ankommen oder im Transit dort vorbeikommen. Jedem Event E in BUF sind die Felder ID(E) und EDGES(E) zugeordnet. Der Beweis für die Korrektheit des Algorithmus findet sich in [Birman87].

Event senden

Zum Senden eines Events E bestimmt der Eventdämon zunächst den spannenden Baum des Eventchannels auf dem der Event gesendet wird und initialisiert EDGES(E) entsprechend der anliegenden Kanten dieses Baums. Wenn E lokal ausgeliefert werden muß, wird E in die lokalen Auslieferungsqueues eingetragen. Zusätzlich kommt E in den Eventpuffer BUF.

Für den Sender ist der Vorgang damit abgeschlossen. Er kann nun so weiterarbeiten, als wäre E an alle Interessenten verteilt.

Der Eventdämon ist für das tatsächliche Versenden eines Events E zuständig. Dafür wird ein Transferpaket erzeugt, das auf allen Kanten in EDGES(E) gesendet wird. Das Transferpaket enthält alle Events E_i aus BUF, von denen E kausal abhängig ist inklusive E selbst. Die E_i werden in aufsteigender Reihenfolge in das Transferpaket kopiert. Nach erfolgreicher Übertragung des Transferpakets auf Kante K wird für jeden Event E_i im Transferpaket K aus EDGES(E_i) entfernt, wenn K dort eingetragen war.

Event empfangen

Empfängt ein Eventdämon ein Transferpaket mit Events E_i, so verfährt er für jeden Event E_i im Paket folgendermaßen. Wenn bereits ein Event mit der ID von E_i in BUF vorhanden ist, ist E_i ein Duplikat und wird verworfen. Andernfalls wird E_i an BUF angehängt. EDGES(E_i) wird entsprechend der Kanten des Baums des Channels initialisiert, ausgenommen die Kante, über die E_i empfangen wurde. Nun prüft der Eventdämon, ob er Teilnehmer des Channels verwaltet, setzt E_i auf die entsprechenden Auslieferungsqueues und entfernt deren Kanten aus EDGES(E_i). Das Weitersenden der empfangenen Events ist im vorigen Abschnitt beschrieben.

Überlegungen zur Optimierung

Der Algorithmus is in der beschriebenen Form noch nicht übermäßig sparsam in Hinblick auf Speicherplatz und Kommunikationskosten.

- Die Transferpakete können zu größeren Einheiten zusammengefaßt werden und verringern so unter Umständen die Anzahl der benötigten Netzwerkpakete.

- In jedem Transferpaket werden Kopien der Events in BUF an die Nachbarn gesendet, egal ob der Nachbar den entsprechenden Event bereits erhalten hat oder nicht. Zur Lösung wird für jeden Event E ein zusätzliches Feld SENT_TO(E) verwaltet, das die Kanten zu Nachbarn enthält, zu denen E bereits gesendet wurde.

- In unserem speziellen Szenario können wir davon ausgehen, daß Events sich sehr schnell über den Baum verteilen und nur noch die Reihenfolgeinformation interessant ist. Das Transferpaket kann in einem ersten Schritt nur die EventIDs enthalten und der empfangende Dämon fordert die Events von bisher unbekannten IDs an. Dies hat den zusätzlichen Vorteil, daß auch ein unzuverlässiger Link für die Kommunikation verwendet werden kann, da Duplikate bereits durch den Algorithmus erkannt werden und verlorengegangene Events spätestens beim Senden des nächsten Events bemerkt werden.

Löschen von Events aus BUF

Der Eventpuffer BUF stellt die Reihenfolgeinformation zur Verfügung, die für die kausalen Abhängigkeiten der Events nötig ist. Bisher haben wir uns keine Gedanken gemacht, wie Events, die bei allen Consumern eines Eventchannels eingegangen sind, aus den diversen BUFs im Baum entfernt werden können. Betroffen sind nur Events E, für die gilt daß EDGES(E) leer ist.

Birman und Joseph beschreiben ein allgemeines Konzept zum Löschen von Events aus BUF, daß unter Verwendung eines assoziativen Speichers eine globale Garbage Collection aller BUFs durchführt. In unserem speziellen Fall können die Events aber auch mit einem einfacheren Mechanismus aus BUF entfernt werden.

Dazu ist es nötig zu bestimmen, welches der neueste Event E_{neu} in allen Eventpuffern BUF eines Channels ist, der bereits an alle Consumer ausgeliefert wurde. Dieser Event und alle kausal davor liegenden Events E_i des gleichen Channels können gelöscht werden. In periodischen Abständen senden die Blattdämonen (d.h. die Dämonen, die nur eine ausgehende Kante zum spannenden Baum haben) die ID ihres lokalen Events E_{neu} in den Baum. Innenliegende Dämonen sammeln die E_{neu} ihrer Kinder, bestimmen daraus und anhand ihres eigenen Puffers BUF ihren E_{neu} und schicken diesen weiter. Am Zentralknoten des Convergecasts wird der globale E_{neu} des Baums bestimmt und daraufhin ein normaler Systemevent gesendet, der den ältesten E_{neu} für den gesamten Baum enthält. Daraufhin können alle Eventdämonen den ältesten E_{neu} und kausal davor liegende Events des Baums löschen, da diese mit Sicherheit nicht mehr referenziert werden.

In die Transportpakete dürfen nur Events kopiert werden, die zwischen E_{neu} und dem zu sendenden Event liegen. Andernfalls könnte nach dem lokalen Löschen von E_{neu} nicht entschieden werden, ob die kausal vorher liegenden Events neu oder nur bereits schon verarbeitet und aus BUF gelöscht sind.

Wenn ein Eventchannel geschlossen wird, können alle Events in BUF, die zu diesem Channel gehören, ebenfalls entfernt werden.

III.2.4. Aufbau eines minimal spannenden Baums

Als grundlegende Kommunikationsstruktur des Restricted-Broadcast-Algorithmus zwischen den Koordinatoren benötigen wir einen spannenden Baum über alle Koordinatoren. Der Baum soll möglichst geringe Kosten aufweisen und später für das Versenden von Verwaltungsevents in Form eines Systemchannels zwischen allen beteiligten Koordinatoren verwendet werden. So kann die Kommunikation zwischen Rechnern des Kernsystems und das Einfügen neuer Koordinatoren verhältnismäßig effizient erledigt werden. Wir gehen im folgenden davon aus, daß der Leser mit grundlegenden Begriffen und Definitionen über Graphen vertraut ist.

Die meisten Algortihmen zur Berechnung eines minimal spannenden Baums basieren auf dem Algorithmus von Kruskal: Es werden disjunkte Fragmente im Graphen gebildet und kombiniert. Ein Fragment besteht aus Knoten und Kanten, die garantiert Teile des spannenden Baums sind. Die Kanten des Fragments bilden dabei über den Knoten des Fragments einen MSB. Der Algorithmus beginnt mit Fragmenten, die nur einen Knoten enthalten. Die Fragmente werden nach und nach zu größeren Fragmenten kombiniert, bis schlußendlich nur noch ein Fragement übrigbleibt, das den MSB über den ganzen Graphen repräsentiert. Eine verteilter Algorithmus dieses Verfahrens stammt von Gallager, Humblet und Spira [Gallager83]. Die obere Grenze für die Zahl der benötigten Nachrichten beträgt $2E + 5N \log_2 N$, wobei N die Anzahl der beteiligten Knoten und E die Anzahl der Kanten zwischen den Knoten bezeichnet.

> *Die theoretische Grenze für die lokale Berechnung eines minimal spannenden Baums liegt bei* N $\log_2$ N. *Der Algorithmus ist also von seiner Komplexität für dünne, planare Graphen, wie sind in der Realität der Rechnernetze vorkommen, bereits sehr dicht an der theoretischen Grenze.*

Der Algorithmus

Wir beschreiben im folgenden das Funktionsprinzip des Algorithmus. Eine ausführliche Beschreibung findet sich in [Gallager83]. Einige Hindernisse bei der Implementierung beschreiben wir im vierten Teil der Arbeit.

Jeder Knoten führt lokal denselben Algorithmus aus. Es gibt keine zentrale Instanz, die den Algorithmus überwacht. Jedes Fragment sucht asynchron zu den anderen Fragmenten die ausgehende Kante mit dem kleinsten Gewicht, die per definitionem Teil des MSB sein muß. Das Fragment versucht dann mit dem Fragment auf der anderen Seite der Kante zu einem neuen, größeren Fragment zu verschmelzen. Wann das geschieht, hängt vom "Level" des Fragments ab, der wiederum von bisherigen Verschmelzungen der beteiligten Fragmente abhängt. Ein Fragment mit einem Knoten ist auf Level 0. Wir zeigen nun, unter welchen Bedingungen Fragmente verschmelzen. Sei ein Fragment F auf Level $L \geq 0$ und sei F' das Fragment auf Level L' auf der anderen Seite der von F ausgehenden Kante mit minimalem Gewicht. Nun gibt es folgende Fälle:

- **L < L'**
  F wird sofort von F' absorbiert. Das erweiterte Fragment F' bleibt auf Level L'.

- **L = L'**
  Wenn F und F' dieselbe ausgehende Kante mit minimalem Gewicht haben, so verschmelzen sie sofort zu einem neuen Fragment F" mit Level L + 1. Die verbindende Kante nennen wir den "core" des Fragments. Da die Gewichte aller Kanten unterschiedlich sind, kann durch Angabe des Gewichts von core ein Fragment eindeutig bezeichnet werden.

- **L > L'**
  Fragment F wartet solange bis Fragment F' einen Level erreicht, so daß einer der anderen Fälle eintritt. Da der Kommunikationsaufwand zum Finden der ausgehenden Kante mit minimalem Gewicht des Fragments proportional zur Größe des Fragments ist, sparen wir durch das Warten im Fragment F' Kommunikationskosten ein.

Der konstruierte MSB ist soweit wie möglich ausgeglichen. Die Wurzel des Baums liegt an der core-Kante des letzten Fragments.

Zwei Dinge müssen noch besonders betrachtet werden. In einem allgemeinen Graphen können wir nicht annehmen, daß das Gewicht zweier Kanten immer unterschiedlich ist. Wir lösen dies, indem die eindeutigen Netzwerkbezeichnungen der anliegenden Knoten (IP-Adressen) mit dem Kantengewicht kombiniert werden. Das tatsächlich zur Berechnung des MSB verwendete Gewicht einer Kante besteht dann aus Kantengewicht + IP-A1 + IP-A2, mit IP-A1 < IP-A2 .

Der Algorithmus enthält keinerlei Fehlererkennung oder Fehlerkorrektur. Wenn einer der beteiligten Knoten während der Abarbeitung des Algorithmus ausfällt, kann kein spannender Baum konstruiert werden. Schlimmer noch, dieser Zustand kann nicht ohne weiteres erkannt werden, wenn nur der Thread, der den Algorithmus ausführt, abbricht. Auch hier sind Timeouts auf Kommunikations- oder Anwendungsebene die einzige Möglichkeit Fehler zu erkennen.

Kriterium zur Neuberechnung des spannenden Baums

Neben der Neuberechnung aufgrund von Fehlerfällen (siehe Kapitel II.8) soll der statische Baum, wie in Kapitel III.2.2 erwähnt, von Zeit zu Zeit neu berechnet werden, um Änderungen in der Netzlast zu berücksichtigen.

Eine Möglichkeit ist, den Baum einfach regelmäßig nach einem gewissen Timeout (zum Beispiel 5-10 Minuten) neu zu berechnen. Dieses Verfahren hat aber deutliche Nachteile:

- Der Baum wird unnötig häufig neu berechnet, wenn die Netzlast sich nicht deutlich ändert, und damit sehr viel Bandbreite verschwendet. Diesen Nachteil kann man abschwächen, indem man die Neuberechnung nur sehr selten durchführt.

- Dann reagiert aber der Baum auf sprunghafte Veränderungen in der Netzlast (zum Beispiel wegen einer Videokonferenz) erst bei der nächsten Neuberechnung. Eine schnellere Reaktion ist aber wünschenswert.

Wir benötigen ein Verfahren, daß dynamisch entscheidet, wann der Baum neu berechnet werden soll. Grundlage dafür ist regelmäßiges Messen der Leitungsqualität und der Verzögerung zu den Nachbarknoten unter Verwendung von PING-Nachrichten, die ohnehin für die Fehlererkennung des Eventmanagers benötigt werden. Die dabei anfallenden Meßdaten werden (unter Umständen kumuliert) lokal gespeichert. Jeder Knoten hat genau eine Kante mit minimalem Gewicht, die Teil des minimal spannenden Baums sein muß und zu den Initialfragmenten im oben beschriebenen Algorithmus gehört.

Wenn sich die Gewichte der anliegenden Kanten so verändern, daß eine andere Kante nun die minimale Kante wird, dann muß sich auch zwingend der MSB ändern. Um ein "Flattern" des Baums wegen ständiger Neuberechnung bei knappen Mehrheiten zu verhindern, führen wir nun noch einen Schwellwert für die Veränderung des Kantengewichts ein. Erst wenn der Schwellwert an einem Knoten überschritten wird, muß der spannende Baum neu berechnet werden.

Damit ist ein eindeutiges und leicht bestimmbares Kriterium zur Neuberechnung von minimal spannenden Bäumen gegeben.

III.2.5. Message Queues

In den im vorigen Kapitel vorgestellten Algorithmen haben wir uns keine Gedanken über die Zuverlässigkeit der Sendeoperationen gemacht. Message Queues (MQs) bieten zuverlässiges Senden und Empfangen von Nachrichten unter Berücksichtigung des Ausfalls von Sender oder Empfänger.

Funktionsprinzip und Eigenschaften

Das Modell von Message Queues wird durch verteilte Applikationen charakterisiert, die durch Nachrichtenaustausch kommunizieren. Die Nachrichten werden indirekt über MQs ausgetauscht. Somit können die beteiligten Applikationen unabhängig voneinander ablaufen. Sie müssen noch nicht einmal gleichzeitig aktiv sein. Die sendende Applikation verwendet eine Schnittstelle der MQ-Verwaltung, um eine Nachricht an eine lokale MQ anzuhängen. Die MQ-Verwaltung ist dafür zuständig, daß die Nachricht zu der gewünschten, eventuell entfernten, MQ transportiert wird, wo sie von der empfangenden Applikation abgeholt werden kann. Auf diese Weise können verteilte Applikationen leichter realisiert werden, indem der Nachrichtenfluß zwischen den Prozessen auf Basis der MQs modelliert wird. Die Entwicklung von verteilten Applikationen wird vereinfacht, indem bestehende Prozesse anders kombiniert werden und somit neue Applikationen entstehen.

Message Queues haben Eigenschaften, die sie für unsere Zwecke interessant erscheinen lassen.

- **Entkopplung von Sender und Empfänger**
  Ein Prozeß, der Nachrichten an die MQ anhängt braucht den Empfänger der Nachrichten nicht zu kennen. Damit ist es möglich, Teilaufgaben einer Arbeit von dafür spezialisierten Prozessen bearbeiten zu lassen, die die Daten aus einer MQ einlesen, verarbeiten und das Ergebnis in eine andere MQ rausschreiben. Das Verfahren ähnelt dem Pipe-Mechanismus von UNIX, mit dem Unterschied, daß MQs jeweils einen Namen und einen bestimmten Platz im Rechner haben.

- **Persistenz**
  Im Gegensatz zu UNIX-Pipes sind MQs dauerhaft vorhanden. Darüberhinaus können Nachrichten in MQs persistent gemacht werden. Das bedeutet, daß Nachrichten beim Anhängen vom MQ-Verwaltungssystem transparent zusätzlich auf stabilem Speicher (Festplatte, Band, ...) gesichert werden und somit auch Ausfälle des Rechners überdauern können.

- **zuverlässige entfernte Kommunikation**
  Unter Verwendung von Message Queues läßt sich leicht zuverlässige, entfernte Kommunikation mit "Once-only"-Semantik realisieren, indem ein Prozeß seine Nachrichten in eine lokale MQ schreibt, die logisch mit einer MQ auf einem anderen Rechner verbunden ist. Wenn der Zielrechner momentan wegen Absturz oder Netzwerkproblemen nicht verfügbar ist, wird die Übertragung der Nachricht vom MQ-Verwaltungssystem

für den Nutzer unsichtbar solange verzögert, bis der Zielrechner mit der entfernten MQ wieder angesprochen werden kann.

Implementierungen

Das Konzept der Message Queues ist seit langem bekannt. Kommerzielle Implementierungen gab es zunächst für IBM-Großrechner. Seit einiger Zeit sind auch Implementierungen für UNIX-Systeme verfügbar. Die bekannteste Produktfamilie kommt von IBM und hört auf den Namen IBM MQseries [MQseries97].

Im Rahmen einer Studienarbeit wurden am IPVR ebenfalls Message Queues implementiert [Rieg96]. Eine herausragende Eigenschaft dieser Implementierung ist, daß sogenannte Eventqueues angelegt werden können. Einer Eventqueue können bis zu 16 entfernte Queues zugeordnet werden [Rieg97]. Damit ist es dann verhältnismäßig einfach, Multicastbäume zu modellieren, und im Eventmanager für den zuverlässigen Versand von Nachrichten zu verwenden. Unter Ausnutzung der Persistenz von MQs sind Events sogar vor Rechnerabstürzen oder Netzausfällen sicher und ein sauberes Wiederaufsetzen nach Abstürzen ist einfacher zu realisieren.

III.2.6. Gruppenkommunikation in Amoeba

Ein Ansatz für sichere Gruppenkommunikation, der zudem auch eine globale Ordnung aller Events sicherstellt, wurde von Kaashoek und Tanenbaum im Amoeba-System implementiert [Kaashoek91]. Amoeba ist ein verteiltes Betriebssystem, daß vorrangig für LANs gedacht ist. Zunächst stellen wir den dort verwendetet Ansatz vor und untersuchen dann, ob und wie dieser Ansatz für sichere Langstreckenkommunikation verwendet werden kann.

Teilnehmer einer Gruppe sind Amoeba-Prozesse auf den einzelnen Rechnern im Netz. Eine Nachricht, die in die Gruppe gesendet wird, erreicht jeden Teilnehmer sobald dieser ReceiveFromGroup() aufruft (Pull-Semantik). Kaashoek und Tanenbaum integrierten zusätzlich Primitive für den Umgang mit Prozessabstürzen in das System.

Das Netzwerk soll möglichst bereits Broadcast oder Multicast unterstützen. Bei Erzeugen der Gruppe wird ein "Sequenzer" bestimmt. Das Versenden einer Nachricht kann auf zwei Arten geschehen. Zum einen kann die Nachricht zunächst an den Sequenzer gesendet werden, der sie mit einer Sequenznummer versieht und sie über den vom Netz zur Verfügung gestellten Broadcastmechanismus in die Gruppe sendet. Zum anderen kann der Sender selber broadcasten. Sobald der Sequenzer einen solchen Broadcast empfängt, numeriert er die Nachricht und broadcastet eine spezielle accept-Nachricht. Erst wenn das accept eintrifft ist, die vorherige Nachricht gültig. Der Sender bleibt so lange blockiert, bis seine eigene Nachricht wieder bei ihm eintrifft. Auf dieser Weise ist die Einhaltung einer globalen Ordnung sehr einfach.

Fehlende Nachrichten werden vom Empfänger anhand der Lücken in der Folge der Sequenznummern spätestens bei Eintreffen der folgenden Nachricht bemerkt und daraufhin nochmals vom Sequenzer angefordert. Der Sequenzer verwendet ein ähnliches Verfahren zum Aufräumen seines "History-Buffers" wie wir es in Kapitel III.2.3 für das Aufräumen des Eventbuffers BUF beschrieben haben. Es werden keine expliziten Bestätigungen für einzelne Nachrichten versendet. Stattdessen werden die Bestätigungen per Piggybacking mit der nächsten Nachricht gesendet. Da alle Nachrichten im System global geordnet sind, können Bestätigungen kumuliert werden und eine Bestätigung gilt somit auch für alle davor empfangenen Nachrichten.

Unter Verwendung dieses Protokolls benötigt man für das Versenden einer global geordneten Nachricht im schlimmsten Fall zwei Broadcasts. Nachteilig ist, daß ein Sender warten muß, bis seine eigene Nachricht wieder bei ihm eintrifft. In einem lokalen Netz ist das keine große Einschränkung, da die Nachricht sehr schnell wieder beim Sender eingeht. In einem Weitverkehrsnetz kann diese Verzögerung den Durchsatz allerdings drastisch verringern.

III.3. Kommerzielle Produkte

In diesem Kapitel stellen wir zwei existierende kommerzielle Produkte kurz vor. Über die Kosten zur Verwaltung der internen Strukturen lassen sich auf Basis des vorliegenden Materials nur wenig Aussagen machen. Das Kapitel soll vielmehr einen Eindruck vermitteln, welche Dienste auf kommerzieller Ebene geboten werden. Die folgenden Abschnitte basieren auf Informationsmaterial der Hersteller.

III.3.1. OrbixTalk

IONA OrbixTalk ist ein Eventmanager, der auf dem Object Request Broker (ORB) Orbix der irischen Firma IONA aufbaut. Orbix ist ein voll CORBA-konform. Dieser Abschnitt orientiert sich am OrbixTalk Whitepaper [Orbix96]. IONA bietet auf Basis von OrbixTalk einen Eventmanager an, der zum in Kapitel III.1 beschriebenen CORBA Event Service kompatibel ist.

OrbixTalk kennt "Talker" (= Supplier) und "Listener" (= Consumer). Talker und Listener unterhalten sich über ein bestimmtes "Topic" (= Eventchannel). Im Gegensatz zu den CORBA Event Services kann ein Objekt bei einem bestimmten Topic entweder nur Talker oder nur Listener sein. Für Antworten auf eingehende Aufträge muß ein Objekt auf einem anderen Topic Talker und der Auftraggeber dort Listener sein. Wahlweise kann der Listener natürlich auch die "normalen" Mechanismen des ORBs wie Remote Method Invocation für die Antwort verwenden.

Listener und Talker verwenden Topic-Namen, um die Informationen zu bezeichnen, für die sie sich interessieren, beziehungsweise die sie versenden. Topic-Namen sind hierarchisch organisiert und ähneln den URLs des World Wide Web. Ein gültiger Name ist beispielsweise "`otrmp//mole/terminierung/aufgabe/infosuche`". otrmp bezeichnet das zu verwendende orbix-eigene Multicastprotokoll. Die folgenden Wörter bezeichnen das Topic. In diesem Fall geht es wohl um die Terminierung von Agenten, die an der Aufgabe "infosuche" teilnehmen.

Im Gegensatz zu CORBA ist die Kommunikation zwischen den Talkern und Listenern immer asynchron. Ein Event wird beim Listener ausgeliefert, sowie er eintrifft.

OrbixTalk verwendet für die Multicast-Kommunikation die IP Multicast Erweiterungen, die in Kapitel III.2.1 ausführlich erläutert sind. Events werden beim Sender für den Fall gespeichert, daß sie wegen Übertragungsfehlern nochmals angefordert werden.

OrbixTalk bietet auch einen sogenannten MessageStore, der von der Funktionalität dem im zweiten Teil beschriebenen Kernsystem entspricht. Durch die Angabe des entsprechenden Protokolls beim Öffnen des Topics werden sämtliche gesendeten Events im MessageStore für die Applikation transparent gespeichert. OrbixTalk garantiert dann die Auslieferung aller Events dieses Topics.

OrbixTalk wird in C++ programmiert, eine Java-Anbindung ist über die CORBA-IDL möglich.

III.3.2. TIBCO Rendezvous

TIBCO Inc. ist eine kalifornische Firma, die mit Rendezvous und dem TIBCO Information Bus (TIB) einfach zu verwendende und dennoch sehr mächtige Middleware-Produkte für das Eventmanagement anbietet. TIB ist der eigentliche Eventmanager, der auf Basis von "Subject-Oriented-Addressing" einen netzwerkweiten "Publish-And-Subscribe"-Mechanismus anbietet. TIB setzt auf UDP auf und verwendet für das Routing zwischen Netzen, im Gegensatz zu Orbix, einen eigenen reliable Broadcast-Routingalgorithmus auf der Basis spannender Bäume. TIB beinhaltet einen vollständigen CORBA ORB.

Nachrichten dürfen bei Rendezvous eine beliebige Länge haben und beliebige Datenstruturen enthalten. Eine Nachricht ist immer einem bestimmten Subject zugeordnet (ähnlich den Eventchannels). Der Namensraum der Subjects ist hierarchisch gegliedert und ein Listener kann unter Verwendung von Wildcards den ganzen Unterbaum eines Subjects abören. TIBCO integriert die Verteilung der Nachrichten in den Eventmanager des je nach Plattform verwendeten Fenstersystems. Eine Applikation erhält also alle Events aus einer Quelle. Für Programme ohne Fensteroberfläche wird ein simpler Eventmanager mitgeliefert. Rendezvous unterstützt C, C++, Java und Perl auf diversen Rechnerplattformen.

Die Architektur unseres Eventmanagers orientiert sich stark an der Rendezvous-Architektur. Auch dort gibt es auf jedem Rechner genau einen Eventdämon der mit anderen Dämonen und den Rendezvous-Routern kommuniziert.

III.4. Bewertung

Die kommerziellen Lösungen bieten grundsätzlich alle eine ähnliche Funktionalität, da sie durchgehend CORBA kompatibel sind oder demnächst sein werden. Der Unterschied liegt meist im zugrundeliegenden firmeneigenen Broadcastsystem. IONA verwendet das eigene Produkt OrbixTalk zur Implementierung des CORBA-kompatiblen Eventmanagers. IBM baut auf der Open-Blueprint-Struktur auf und TIBCO verwendet den eigenen TIBCO-Information-Bus zur Informationsübermittlung innerhalb von Rendezvous. Allen Lösungen gemeinsam ist, daß sie Geld kosten. Das Agentensystem MOLE und die Software, die zu seinem Betrieb notwendig ist, sollen frei erhältlich sein. Aus diesem Grund soll für die Implementierung des Eventmanagers auch nur freie Software verwendet werden. Das ist keine wissenschaftliche, sondern eine politische Entscheidung der Agentengruppe.

Bestünde diese Einschränkung nicht, so wäre Orbix aufgrund der guten CORBA-Implementierung oder TIBCO Rendezvous wegen der einfachen und leicht verständlichen Programmierschnittstelle zu favorisieren.

Für die Implementierung wählen wir daher eine Lösung auf Basis der vorgestellten Basic Building Blocks und unter Verwendung der inkrementellen Multicastprotokolle RB und MOMP. Wichtig ist in diesem Zusammenhang die Wahl des grundlegenden Kommunikationsmechanismus. Die Kommunikation der Eventdämonen mit dem Koordinator stellt andere Anforderungen als die Kommunikation der Koordinatoren untereinander. Für ersteres bieten sich UDP-Nachrichten an, die in einem bus-basierten, lokalen Netz mit Broadcast oder lokalem Multicast versendet werden sollten. Da sich die Kommunikationswege zwischen den Koordinatoren eher selten ändern, sollte für diese eine verbindungsorientierte Kommunikationform verwendet werden. Message Queues sind im Vergleich zu TCP zwar das besser passende Konzept, aber leider ist die Anbindung der am IPVR entwickelten Message Queues an das MOLE-System schwierig und aufgrund der extrem starken Ausrichtung an UNIX-IPC-Mechanismen nur sehr umständlich auf andere von MOLE unterstützte Betriebssysteme wie Windows NT portierbar.

IV. Implementierung

In diesem Teil der Diplomarbeit beschreiben wir die Implementierung des Eventmanagers und seine Anbindung an das Agentensystem MOLE.

IV.1. Entwurf der Kommunikationsstrukturen

In diesem Kapitel stellen wir den Entwurf der Kommunikationsstrukturen vor. In Abbildung IV.1 ist die die Kommunikationsstruktur als Schichtenmodell dargestellt. Wir gehen auf die einzelnen Komponenten später noch genauer ein.

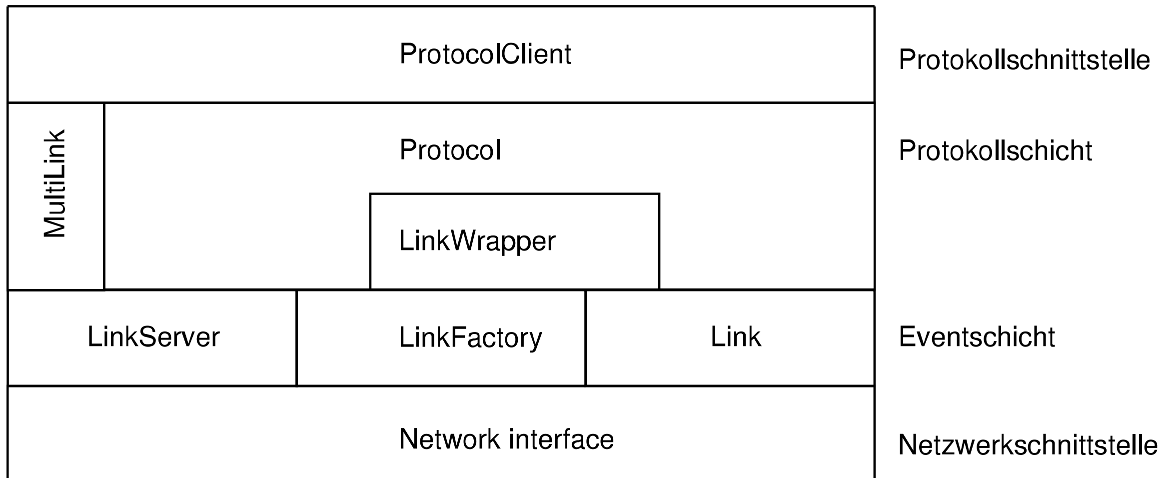


Abb. IV.1: Schichtenmodell der Kommunikationsstruktur des Eventmanagers.

Die Eventschicht baut auf der in Java vorhandenen Netzwerkschnittstelle auf und bildet zunächst die dort vorhandenen Netzwerkdienste TCP, UDP und IP-Multicast auf eine Eventschnittstelle ab. Der LinkServer verwaltet die Verbindungen des entsprechenden Netzwerkdienstes zu den Kommunikationspartnern. Eine neue Verbindung ("Link") wird unter Verwendung einer Linkfactory hergestellt, die zusammen mit dem LinkServer dafür sorgt, daß die Verbindung ordnungsgemäß geöffnet werden kann. Die UDPLinkFactory prüft beispielsweise nach, ob mit dem angegebenen Nachbarn überhaupt auf dem gewünschten Port kommuniziert werden kann. Das zu implementierende Protokoll kann seine Dienste selbst wieder in einen Link verbergen, der je nach Konfiguration einen der Linkdienste aus der Eventschicht verwendet. Dadurch ist es möglich, ein und dasselbe Protokoll über unterschiedliche Netzwerkdienste zu fahren, da sie alle die gleiche Eventschnittstelle anbieten. Ebenso kann unter Verwendung der LinkWrapper-Klasse, ein funktionell erweiterter Netzwerkdienst implementiert und verwendet werden. Das Protokoll kann wahlweise in einem Link (für sehr einfache Protokolle wie PING) oder in einer Unterklasse von MultiLink und eventuell Verwendung einer eigenen Linkklasse implementiert werden (Gallager, Restricted Broadcast). Der MultiLink vereinfacht die Verwaltung von Links zu mehreren Nachbarn. Verwendet und gesteuert wird das ganze System von den einzelnen Protokoll-Clients, die auch die Schnittstelle zum Rest des Systems zur Verfügung stellen.

Der Eventmanager wurde mit Java 1.1 implementiert. In den folgenden Kapiteln gehen wir auf die einzelnen Komponenten näher ein.

IV.1.1. Der Link

Pro Eventchannel soll ein eigenes Eventprotokoll verwendet werden können. Gleichzeitig soll aber jedes Protokoll über dieselbe Programmschnittstelle angesprochen werden können. Um die Implementierung des Protokolls unabhängig von der Implementierung der Schnittstellen zu halten, verwenden wir als abstraktes Kommunikationsmittel den "Link". Ein Link ist konzeptionell eine Verbindung zwischen zwei (in einem Ausnahmefall auch mehreren) Prozessen, die gewisse Dienstmerkmale garantiert und über die Events gesendet werden. Ein Link bietet die folgende Schnittstelle an:

- Link öffnen
- Link schließen
- Event senden
- Eventconsumer für Eventempfang registrieren

Hinzu kommen noch einige Hilfsmethoden, die wir hier vernachlässigen wollen. Sie sind, wie das gesamte Eventmanagementsystem, online dokumentiert und über die MOLE-Projektseite verfügbar [MOLE97].

Links bestehen zwischen Prozessen auf (normalerweise) unterschiedlichen Rechnern und bieten unterschiedliche Dienstqualitäten an. In einem ersten Schritt implementierten wir die physikalischen Links für die Eventschicht: TCPLink, UDPLink und MulticastLink, die lediglich eine Abbildung der Eventschnittstelle auf die im Netz verwendete byteorientierte Form darstellen und genau die gleichen Dienstmerkmale wie die zugrundeliegenden Protokolle bieten. Durch die einheitliche Schnittstelle können nun verschiedene Linkimplementierungen für ein höheres Protokoll verwendet werden.

Beispielsweise kann so in einer Anwendung der bisher verwendete UDPLink einfach durch Austausch eines Konstruktors durch einen ReliableUDPLink ersetzt werden, der zusätzlich Timeoutmechanismen für Wiederholungssendungen von Events sowie Duplikatserkennung implementiert. ReliableUDPLink verwendet einen UDPLink für die Kommunikation und implementiert nur die zusätzlichen Funktionen. Dieses Vorgehen bietet den Vorteil, daß ohne große Änderung in einer höheren Schicht des Eventmanagers, die Kommunikationsstrukturen und Protokolle ausgetauscht werden können. Das ist besonders hilfreich für Messungen und die Implementierung von neuen Eventprotokollen.

Der einzige Link zwischen gleichzeitig mehren Prozessen, ist der LocalBroadcastLink. Wir "mißbrauchen" dafür den IP-Multicast, indem als Zieladresse 224.0.0.1 (aller Rechner in diesem Netz) verwendet wird. Alternativ könnte man auch die echte lokale Broadcastadresse des Netzes verwenden (in der Abteilung VS beispielsweise 129.69.210.255). Leider akzeptiert das Java-Laufzeitsystem diese Adresse nicht als Zieladresse eines UDP-Pakets. Da die Broadcastadresse von der Subnetz-Maske des LANs abhängt, ist ihre Bestimmung aus Java heraus nicht möglich und muß in einer Konfigurationsdatei spezifiziert werden.

IV.1.2. Die Linkfactory

Der Link ist ein logisches Konstrukt, das durch je ein Linkobjekt auf jeder Seite des Links instanziiert wird. Linkobjekte sollen nicht direkt durch Anwendungsprogramme angelegt werden können, da sie diverse Parameter haben, die vor einer Instanziierung geprüft werden müssen (beispielsweise, ob der angegebene Nachbar überhaupt existiert und erreichbar ist). Für jeden Linktyp existiert daher eine Linkfactory, deren einzige Aufgabe es ist, die für die Instanziierung nötigen Parameter (zum Teil unter Absprache mit dem Nachbarn über den Systemkanal) zu prüfen und das Linkobjekt zu erzeugen. Damit kann garantiert werden, daß ein Linkobjekt immer ein passendes Gegenstück auf "der anderen Seite" hat. Die Schnittstelle einer Linkfactory ist sehr einfach.

- Parameter prüfen und zugehörigen Link erzeugen
- nur Parameter prüfen

Alternativ könnte man auch folgendermaßen vorgehen: Das Linkobjekt wird angelegt und mit den entsprechenden Parametern attributiert. Erst dann wird der Link tatsächlich geöffnet. Das Verfahren hat aber den Nachteil, daß der Programmierer unter Umständen mehrere verschiedene Attributierungsmethoden des Links aufrufen muß, bevor der Link geöffnet wird und die Attribute bei jeder Methode des Links zuerst auf Korrektheit geprüft werden müßten.

IV.1.3. Der Linkserver

Um eine einheitliche Eventschnittstelle zu erhalten, verwenden wir das Konzept der Links und gleichen so die unterschiedlichen Semantiken von TCP-Sockets und UDP-Sockets aus. Ein Linkserver verwaltet jeweils alle angemeldeten Links seines Typs zu den Nachbarn (UDP/TCP- und MulticastLinks). Linktypen höherer Protokolle verwenden die Klasse LinkWrapper, um ihre erweiterte Funktionalität zu implementieren. Die Linkfactory der Links verwendet den entsprechenden LinkServer um die Linkobjekte anzulegen. Der LinkServer bietet folgende Schnittstelle.

- Link des entsprechenden Typs zum angegebenen Nachbarn anlegen beziehungsweise eine Referenz auf den bereits bestehenden Link zurückgeben
- Link schließen

Der physikalische Link wird natürlich erst geschlossen, wenn sich der letzte Nutzer abmeldet.

IV.1.4. Der MultiLink

Links beschreiben, wie Events zwischen zwei Prozessen versendet werden. Da wir aber auf Baumstrukturen arbeiten, wird ein Event meist entweder zu einem bestimmten Nachbarknoten gesendet oder zu allen Nachbarknoten. Der MultiLink unterstützt den Programmierer bei der Verwaltung der Nachbarn. Er besitzt die folgende Schnittstelle.

- Link zu Nachbar anlegen
- Link zu Nachbar löschen
- Event an alle Nachbarn senden
- Event an bestimmten Nachbar senden
- Event an alle Nachbarn, außer einem bestimmten, senden

Die letzte Funktion kann verwendet werden, um einen empfangenen Event an alle Nachbarn weiterzuleiten, außer dem von dem der Event empfangen wurde ("Flooding").

Beim Anlegen eines MultiLinks muß der zu verwendende Linktyp durch Angabe der zugehörigen Linkfactory spezifiziert werden. Weiterhin muß angegeben werden, von wo der MultiLink seine Events beziehen soll (LinkServer oder andere Eventsupplier).

IV.1.5. Events und Event-IDs

Events werden jeweils in einer eigenen Klasse implementiert, die von events.basics.NetEvent abgeleitet werden muß. Jeder Anwendung wird administrativ ein Zahlenraum für die Major-Eventnummer zugeordnet (16bit). Die Minor-Eventnummern können von der Anwendung beliebig verwendet werden (8bit). Zum Versenden von Events muß die Eventklasse eine Abbildung ihrer Daten auf die Strukturen von NetEvent liefern. Das Kommunikationssubsystem arbeitet nur mit NetEvents, die bei Auslieferung an die Anwendung nach ihrer Major und Minor-Nummer in die entsprechende Subklasse konvertiert werden. Jeder Event enthält einen Absender und den Kurznamen des Eventchannels auf dem er gesendet wird. Anhand des sendenden Rechners und des Kurznamens wird der Event an den richtigen Link weitergeleitet.

Event-IDs bestehen, wie im zweiten Teil der Arbeit beschrieben, aus der IP-Addresse des Hosts, der Portnummer des physikalischen Links, Uhrzeit und laufender Nummer. Leider hat Java einen Fehler (oder sollte man besser sagen ein Feature?) in der Implementierung des Aufrufs InetAddress.getLocalHost(). Es ist von der TCP/IP-Konfiguration und dem Betriebssystem abhängig, welche IP-Addresse dieser Aufruf liefert. Auf den SPARC-Stationen der Abteilung Verteilte Systeme erhält man bis Solaris 2.4 als Ergebnis 127.0.0.1, also die Loopback- Adresse von IP. Ab Solaris 2.5 erhält man eine echte IP-Addresse des Rechners. Bei Rechnern mit mehreren Netzschnittstellen (zum Beispiel für Ethernet und ATM) diejenige der Standardschnittstelle. Andere Betriebssysteme verhalten sich ähnlich uneinheitlich. Unter Java 1.0 hatte der Programmierer keine Chance herauszufinden auf welchem Host sein Programm läuft. Damit wäre eine eindeutige Event-ID nur per Handkonfiguration des Eventdämonen auf jedem einzelnen Rechner realisierbar.

Auch unter Java 1.1 ist es nicht ganz einfach an die tatsächliche IP-Adresse des Rechners zu kommen. Die Socketklassen bieten zwar jetzt die Möglichkeit abzufragen, über welche Netzschnittstelle sie ihre Daten senden. Legt man aber ein Socket ohne explizite Angabe der Schnittstelle an und fragt dann die verwendete Schnittstelle ab, so erhält man die IP-Addresse 0.0.0.0 . Ein an den selben Host gesendetes UDP-Paket enthält als Absender 127.0.0.1. Eine entsprechende TCP-Verbindung kommt zu dem selben Ergebnis. Erst ein UDP-Paket, das in eine temporäre, lokale IP-Multicastgruppe gesendet wird, enthält die

tatsächliche IP-Adresse des Rechners als Absender. Für diesen Trick benötigt man aber allerdings eine IP-Schicht mit Multicast-Unterstützung (IGMP Level 2).

Dieses Verhalten ist nachvollziehbar, wenn man bedenkt, daß erst das Multicastpaket tatsächlich den Rechner verläßt. Alle anderen Verbindungen werden intern von der IP-Schicht behandelt und nicht auf dem lokalen Netz gesendet.

IV.2. Die Protokollschnittstelle

Wir beschreiben nun kurz die diversen Klassen, die das System steuern und die Schnittstelle zu den unterschiedlichen Protokollen bilden.

IV.2.1. Der PingClient

Der PingClient ist für das regelmäßige "anpingen" von Nachbarknoten zuständig. Der PingClient eines Koordinators bestimmt so die Zeit, die ein Ping für den Weg zum Ziel und zurück benötigt. In einem lokalen Netz empfängt der PingClient die regelmäßigen Reports des Koordinators und bestimmt so, ob in seinem lokalen Netz ein Koordinator vorhanden ist. Pings bestehen aus normalen UDP-Paketen.

Eigentlich wollten wir die Entfernungen zu den Nachbarknoten bei den Netzwerkbasisdiensten (beispielsweise mit SNMP beim Router) abfragen. Leider ist das aufgrund von Zugangsbeschränkungen zu fremden Routern nicht möglich. Darüberhinaus kennt im Internet kein Router alle Zielrechner. Stattdessen wird unter Verwendung des "default"-Eintrags in der Routingtabelle das IP-Paket zu einem anderen Router weitergereicht, unter der Hoffnung, daß dieser besser weiß, wo das Paket hingesendet werden muß. Wir hätten zusätzlich auch noch die default-Einträge der Routingtabellen verfolgen müssen, was im Endeffekt viel teuerer gekommen wäre, als regelmäßige Pings. Darüberhinaus können wir mit Pings über die Verlustrate von UDP-Paketen auch die Qualität der Verbindung abschätzen.

IV.2.2. Der Channel-Client

Der Channel-Client ist in jedem Eventmanager-Prozeß vorhanden und für die Verwaltung der Eventchannels zuständig. Für jeden Channel gibt es einen Administrator, der für die Behandlung von Fehlern zuständig ist. Beispielsweise wird so der PingClient informiert, wenn ein Ping von einem bisher unbekannten Nachbarn eintrifft.

IV.2.3. Der StaticMSB-Client

Der StaticMSB-Client (SMSBClient) ist in jedem Koordinator vorhanden und berechnet auf Basis der Nachbarschaftsbeziehungen des PingClients den spannenden Baum über alle Koordinatoren. Dafür mußte der in Kapitel III.2.4 vorgestellte Algorithmus erweitert werden.

Da die PingClients der einzelnen Koordinatoren unabhängig voneinander arbeiten, kann es vorkommen, daß sie für dieselbe Kante unterschiedliche Gewichte berechnen. Die SMSBClients der einzelnen Koordinatoren müssen sich deshalb zunächst über das Gewicht der verbindenden Kanten einig werden. Dies geschieht dadurch, daß jeder SMSBClient sein Gewicht zum Nachbarn schickt und das arithmetische Mittel über die zwei Gewichte bildet. Damit steigt die Zahl der benötigten Nachrichten um $2E$.

Nach Beendigung des Algorithmus müssen alle Knoten informiert werden, daß der Baum nun gültig ist. Nur die zwei Knoten, die an der core-Kante des endgültigen Fragments liegen, halten an. Alle anderen Knoten bleiben aktiv und warten auf weitere Nachrichten. Wir beheben das Problem, indem die zwei haltenden Knoten jeweils in ihren Unterbaum die Nachricht senden, daß der Baum nun fertig konstruiert ist. Zwischenknoten leiten diese Nachricht bis zu den Blättern weiter. Damit steigt die Zahl der benötigten Nachrichten um N-2. Die obere Grenze für die Zahl der benötigten Nachrichten liegt somit bei $4E + 5N \log_2 N + N$.

IV.2.4. Der RB-Client

Auch der RB-Client ist im Koordinator ansässig und für die Berechnung der dynamischen Channels auf Basis des spannenden Baums über alle Koordinatoren zuständig. Seine Aufgabe ist zweigeteilt. Zum einen verwaltet er alle lokal bekannten Channels und leitet Nachrichten dieser Channels weiter, wie in Kapitel III.2.2 beschrieben. Zum anderen kommuniziert er per lokalem Broadcast mit den in seinem Netz vorhandenen Eventdämonen und verarbeitet deren Anforderungen und Datennachrichten.

Zur Realisierung der ChannelRec-Struktur verwenden wir eine Unterklasse von MultiLink. Ein weiterer MultiLink wird für die Beschreibung der Nachbarn auf Basis der vom SMSBClient berechneten Nachbarschaftsbeziehungen verwendet. Zur Kommunikation mit den im lokalen Netz vorhandenen Teilnehmern verwenden wir den MulticastLink. Zur Kommunikation der RBClients in den Eventdämonen mit MOLE verwenden wir die in Kapitel IV.3.1 beschriebenen Schnittstellen.

IV.2.5. Der MOMP-Client

Jeder Dämon betreibt einen MOMP-Client, der bei Migrationen von Agenten das in Kapitel III.2.2 beschriebene Mobile-Object-Multicast-Protocol durchführt. Dies ist, wegen der Vorraussetzungen für MOMP, natürlich nur für Eventchannels möglich, die Events zumindest kausal ordnen. Die Abkürzung während einer Migration wird direkt bei dem MultiLink des entsprechenden Eventchannels eingetragen, aber vom MOMP-Client verwaltet.

IV.3. Anbindung an MOLE

In diesem Kapitel beschreiben wir kurz die benötigten Schnittstellen für die Anbindung des Eventmanagers an MOLE.

IV.3.1. EventDämon und lokaler Eventmanager

Der Eventdämon des Hosts und der lokale Eventmanager des Agentensystems kommunizieren über eine lokale TCP-Verbindung. Der Dämon übernimmt die Rolle des TCP-Servers. Die Events werden in der durch den Eventdämon festgelegten Reihenfolge an das Agentensystem ausgeliefert. Der lokale Eventmanager benötigt also keine Mechanismen für das Ordnen von Events, sondern ist lediglich für das Herstellen der Verbindung zum Eventdämon und das Weiterleiten von Anfragen und Events zuständig. Die Events des Agentensystems werden vom lokalen Eventmanager in ihre serialisierte Form umgesetzt und erst vom lokalen Eventmanager auf einer Zielengine deserialisiert.

Jedes Kommando an den Eventdämon ist ebenfalls eine Unterklasse von NetEvent und enthält eine laufende Nummer. Der Eventdämon quittiert die Ausführung von Kommandos mit einem OKEvent und der laufenden Nummer des Kommandos.

- **create-channel(Channel, Attributes)**
  Ein Objekt will einen neuen Eventchannel anlegen und ihm beitreten. Meist wird dies der Group-Coordinator einer Agentengruppe sein. Es kann Agentengruppen geben, deren Eigenschaften und Verhalten nach Kapitel II.2 bereits während der Entwicklung bekannt ist. Beim Anlegen des Eventchannels wird dem Eventmanager daher ein Satz von Attributen übergeben, der das voraussichtliche Verhalten der Agentengruppe beschreibt. Der Aufruf schlägt fehl, wenn der benannte Eventchannel bereits existiert.

- **delete-channel(Channel)**
  Der benannte Eventchannel soll gelöscht werden.

- **query-channels()**
  Liefert Informationen über alle Eventchannels, die der Eventdämon kennt. Die Informationen werden vom ChannelClient geliefert.

- **lookup-channel(Channel)**
  Prüft, ob der benannte Channel existiert und liefert soweit möglich Informationen über den Channel.

- **join-channel(Channel)**
  Ein Objekt will Teilnehmer des benannten Eventchannels werden. Es ist irrelevant, ob dies ein Supplier oder Consumer ist, da in unserem Modell auch Supplier Gruppenmitglieder sein müssen (im Gegensatz zu CORBA oder IP-Multicast).

- **leave-channel(Channel)**
  Ein Objekt will den benannten Eventchannel verlassen. Dieses Kommando kann, wie join-channel auch, Änderungen in der Kommunikationsstruktur des Eventmanagers auslösen und ist daher verhältnismäßig teuer.

- **move(toHost, Channels)**
  Ein Objekt will auf einen anderen Rechner wechseln, aber weiterhin Mitglied der benannten Eventchannels bleiben. Hier tritt der MOMP-Algorithmus in Aktion.

- **send(Event, Channel)**
  Wird in beiden Richtungen verwendet. Der lokale Eventmanager sendet lokal erzeugte Events der Anwendungen zum Eventdämon und dieser über das Netz empfangene Events zum lokalen Eventmanager. Lokal erzeugte Events dürfen nicht sofort lokal ausgeliefert werden, sondern erst, wenn sie vom Eventdämon empfangen werden, da erst dann eventuelle Reihenfolgebedingungen garantiert sind.

IV.3.2. Lokaler Eventmanager und MOLE

Dies ist die eigentliche Schnittstelle des Eventmanagers zum Agentensystem. Sie bildet das Kommunikationsprotokoll zwischen Eventdämon und lokalem Eventmanager auf eine objektorientierte Schnittstelle ab. Die im vorigen Abschnitt beschrieben Aufrufe werden von einem Eventchannel-Objekt implementiert, das vom lokalen Eventmanager erzeugt und verwaltet wird. Darauf aufbauend können nun Anwendungen für das Agentensystem geschrieben werden, die den in dieser Arbeit vorgestellten Eventservice verwenden.

Das Eventchannelobjekt implementiert die Kommandos create-channel, join-channel, leave-channel und send. Die restlichen Kommandos werden vom lokalen Eventmanager implementiert.

IV.4. Ausblick

In dieser Diplomarbeit haben wir die Grundlagen für einen Eventmanager entwickelt, der für mobile Supplier und Consumer geeignet ist und darüberhinaus die Events pro Eventchannel zuverlässig und kausal geordnet ausliefern kann. Pro Eventchannel kann ein eigenes Eventprotokoll verwendet werden. Die Verteilungsmechanismen können auf einfache Weise umkonfiguriert und um andere Eventprotokolle erweitert werden.

Wir haben die Anbindung an das Agentensystem MOLE beschrieben. Damit können nun auf Basis von Events Agentengruppen und Verfahren zur Synchronisierung von Agenten in weiterführenden Arbeiten behandelt und implementiert werden.

Natürlich haben wir auch einige Bereiche auf dem weiten Feld des Eventmanagements aus Zeitgründen nur gestreift oder ganz weggelassen. Im folgenden eine Auswahl von Themen, die es wert sind, näher betrachtet zu werden.

Der hier beschriebene Eventmanager verteilt die Events unter Verwendung genau eines minimal spannenden Baums. Events werden nur auf den Kanten dieses spannenden Baums weitergeleitet. Interessant sind hier Untersuchung und Einbindung anderer Verteilungsstrategien. Senderbased-Trees sind eine Kommunikationsstruktur, die die minimal mögliche Verzögerung bei der Auslieferung von Events an alle Empfänger garantiert. Es könnten Algorithmen untersucht werden, die ohne den Overhead von IP-Multicasts auskommen. Eine Erweiterung des Restricted-Broadcast-Algorithmus kann auch Routen, die nicht auf dem statischen Baum liegen, für den dynamischen Baum auswählen. Als Ausgangspunkt für weitere Arbeiten bietet sich hier der im Anhang vorgestellte Nearest-Insertion-Algorithmus an.

Die Implementierung weiterer Eventprotokolle ermöglicht eine sinnvolle Auswertung der Attribute von Agentengruppen, die beim Öffnen eines Eventchannels angegeben werden können. Wir haben einige Attribute in dieser Arbeit definiert (Kapitel II.2) , aber ansonsten nicht weiter verwendet.

Die Fehlertoleranz des Systems kann noch gesteigert werden. Der Ausfall von Prozessen des Eventmanagers wird bestenfalls bemerkt und nur zum Teil behandelt. Es gibt bisher auch keine sinnvollen Mechanismen, um ausgefallene Threads (!) wieder neu aufzusetzen.

Ein weiteres Gebiet ist die Verwaltung von Agentengruppen und des Namensraums von Eventchannels. Darüberhinaus kann die Konfiguration des gesamten Eventmanagers vereinfacht und möglicherweise sogar größtenteils dynamisch erledigt werden. Gerade im Verwaltungsbereich hat auch MOLE noch einige Schwachpunkte, die dann in einem ganzeheitlichen Ansatz miterledigt werden könnten.

Sicherheit ist in dieser Arbeit kein großes Thema gewesen. Man kann sich noch viel mehr Gedanken machen, ob Autorisierung und Authentifizierung von Suppliern und Consumern nicht doch besser vom Eventmanager erledigt werden sollte. Insbesondere dann, wenn man die Anwendungsdomäne MOLE verläßt.

Sobald das ständig aktive Kernsystem von MOLE an der Fakultät Informatik installiert ist und funktioniert, rückt die Realisierbarkeit von persistenten Events in greifbare Nähe. Es sollte dann zunächst untersucht werden, für welche Zwecke persistente Events in Agentensystemen tatsächlich benötigt werden.

V. Anhänge

In den Anhängen finden sich das Literaturverzeichnis, eine Übersicht wichtiger Begriffe und Abkürzungen, sowie eine Kurzeinführung in logische Uhren.

Weiterhin sind hier die verwendeten Algorithmen in Pseudocode dargestellt, soweit sie im Hauptteil der Arbeit erläutert werden.

A. Logische Uhren

Wie in der Einführung erwähnt, genügt es in vielen Fällen bereits zu wissen, in welchem zeitlichen Verhältnis Events zueinander stehen, in welcher Reihenfolge zwei abhängige Events aufgetreten sind. Zur Spezifikation dieser Relation in einem System unabhängiger Prozesse (und somit auch unterschiedlicher Rechner) geht Lamport in [Lamport78] einen sehr pragmatischen Weg. Jeder Prozeß (in unserem Fall also jeder Supplier und Consumer) besteht aus einer Folge von Ereignissen. Lamport definiert "logische" Uhren, die die zeitliche Einordnung dieser Ereignisse erlauben.

Zunächst definiert Lamport die "happened before"-Relation "$\rightarrow$" , die folgendes aussagt:

1. a und b sind Ereignisse im selben Prozeß und a erscheint vor b, dann gilt $a \rightarrow b$.
2. Wenn a das Senden einer Nachricht in einem Prozeß und b das Empfangen derselben Nachricht in einem anderen Prozeß ist, dann gilt $a \rightarrow b$.
3. Wenn $a \rightarrow b$ und $b \rightarrow c$, dann gilt $a \rightarrow c$.
4. Zwei Ereignisse a und b, $a \neq b$ heißen gleichzeitig, wenn weder a $\rightarrow b$ noch $b \rightarrow a$ gilt.

Mit anderen Worten: Es gilt $a \rightarrow b$, wenn a eine Auswirkung auf b haben kann.

Es ist leicht einzusehen, daß "$\rightarrow$" eine Halbordnung über die Events aller beteiligten Prozesse definiert.

Ganz abstrakt gesehen ist eine Uhr nichts weiter als die Zuweisung einer Nummer zu einem Ereignis, wobei die Nummer als die Zeit bezeichnet wird, an der das Ereignis aufgetreten ist.

Nun definieren wir die Uhr C_i für jeden Prozeß P_i als eine Funktion, die eine Nummer $C_i\langle a\rangle$ dem Ereignis a zuweist. Das ganze System aller Uhren wird durch die Funktion C beschrieben, die zu jedem Ereignis b die Nummer $C\langle b\rangle$ zuweist, mit $C\langle b\rangle = C_i\langle b\rangle$ für jeden Prozeß P_i. Man beachte, daß die C_i in keiner Relation zur physischen Zeit stehen und daher logische Uhren genannt werden. Die Uhren C_i lassen sich durch einfache Zähler realisieren.

Die Korrektheitsbedingung für die Uhren lautet nun:

- Für alle Ereignisse a, b gilt: wenn $a \rightarrow$ b , dann $C\langle a\rangle < C\langle b\rangle$

Diese Bedingung läßt sich mit drei Regeln einhalten (analog zur Definition von "$\rightarrow$"):

- Jeder Prozeß P_i erhöht C_i zwischen zwei direkt aufeinander folgenden Ereignissen.

- Wenn Ereignis a das Senden einer Nachricht m von Prozeß P_i ist, dann enthält m den Zeitstempel $T_m = C_i\langle a\rangle$

- Beim Empfang der Nachricht m setzt Prozeß P_j seine Uhr C_j größer oder gleich ihrem aktuellem Wert und größer als T_m.

Damit können kausale Abhängigkeiten zwischen Prozessen beschrieben werden, wenn jede Aktion, die eine Auswirkung auf andere Prozesse haben kann, von der andere Prozesse also kausal abhängig sind, durch eine Nachricht an die anderen Prozesse bekanntgegeben wird.

Logische Uhren ermöglichen die Bestimmung einer Rangfolge von Nachrichten, die dem System bekannt sind. Da in den Folgen der Zähler immer wieder Sprünge enthalten sind (verursacht durch das Eintrffen einer Nachricht mit höherem Zählerwert), kann nur auf Basis von logischen Uhren das Fehlen einer bestimmten Nachricht nicht erkannt werden.

B. Algorithmen für dynamischen Multicast

Dieser Abschnitt enthält die modifizierten Algorithmen für dynamischen Multicast.

Restricted Broadcast

Dieser Algorithmus wird für die Kommunikation zwischen den Koordinatoren verwendet. Eine ausführliche Beschreibung findet sich in Kapitel III.2.2 . Der folgende Code wird vom Koordinator auf Knoten *i* ausgeführt.

```
// Definition of records

class RBChannelRec
{
     int            status;
     Coordinator[] neighbourList;
}

// Restricted Broadcast:    executed at coordinator i

case local operation of

join channel M:
     if RBChannelRec(M) exists then
          RBChannelRec(M).status := member;
     else
          allocate RBChannelRec(M);
          RBChannelRec(M).status := member;
          for each node j in Parent(i) ∪ Child(i)
               send join-request to j;
               if reply contains participant then
                    add j to RBChannelRec(M).NeighbourList;
               end if
          next
     end if

delete from channel M:
     if Size(RBChannelRec(M).NeighbourList) > 1 then
          RBChannelRec(M).status := non-member;
     else
          if Size(RBChannelRec(M).NeighbourList) = 1 then
               send delete-request to node
                                        in RBChannelRec(M).NeighbourList;
          end if
          deallocate RBChannelRec(M);
     end if

send event m to channel M:
     if RBChannelRec(M) exists then
          deliver m to local members;
          send a copy of m to each node in RBChannelRec(M).NeighbourList;
     else
          find a participant and send m to it
     end if
```

Dieser Code wird bei Empfang einer Nachricht von Knoten *j* vom Koordinator auf Knoten *i* ausgeführt.

```
// Restricted Broadcast:     executed from coordinator at node i when
//                           a event is received from node j

case event type of

data event m:
     deliver m to local members;
     send copies of m to all nodes in
                                  RBChannelRec(M).NeighbourList except j;

delete-request:
     remove j from RBChannelRec(M).NeighbourList;
     if RBChannelRec(M).status = non-member and
          Size(RBChannelRec(M).NeighbourList) = 1 then
          send delete-request to node in RBChannelRec(M).NeighbourList;
          deallocate RBChannelRec(M);
     end if

join-request:
     if RBChannelRec(M) exists then
          add j to RBChannelRec(M).NeighbourList;
          send a event containing participant to j;
     else
          send join-request to all broadcast neighbour nodes except j;
          if at least one node returns participant then
               allocate RBChannelRec(M);
               RBChannelRec(M).status := non-member;
               add nodes that return participant to
                                           RBChannelRec(M).NeighbourList;
               return participant to j;
          else
               return non-participant to j;
          end if
     end if
```

Mobile Object Multicast Protocol

Wird auf Ebene der Eventdämonen zur Kommunikation zwischen Eventdämonen und Koordinatoren, sowie zur Kommunikation zwischen Eventdämonen verwendet (Shortcuts). Der folgende Code wird auf Knoten *i* ausgeführt.

```
// Definition of records

class MOMPChannelRec
{
     int            status              := non-member;
     int            localMembers        := 0;
     Neighbour[]    shortcutList;
     Neighbour[]    remoteShortcutsFrom;
     EventID[]      shutdownEventID;
}

// Mobile Object Multicast Protocol:  executed at node i

case local operation of

join channel M:
     if MOMPChannelRec(M) does not exist then
          allocate MOMPChannelRec(M);
          send join-channel(M) to coordinator;
          MOMPChannelRec(M).status := member;
     end if
     MOMPChannelRec(M).localMembers++;

delete from channel M:
     MOMPChannelRec(M).localMembers--;
     if MOMPChannelRec(M).localMembers = 0 and
        MOMPChannelRec(M).shortcutList is empty
          send leave-channel to coordinator;
          send clear-shortcut to all hosts
                              in MOMPChannelRec(M).remoteShortcutsFrom
          deallocate MOMPChannelRec(M);
     endif

send event m to channel M:
     if MOMPChannelRec(M) exists then
          deliver m to local members;
          send a copy of m to each node in MOMPChannelRec(M).shortcutList
     end if
     send m to coordinator;

migration of object using channel M to node k:
     send shortcut-added(M) to k;
     add k to MOMPChannelRec(M).shortcutList;


end case
```

Dieser Code wird bei Empfang einer Nachricht von Knoten *j* auf Knoten *i* ausgeführt.

```
// Mobile Object Multicast Protocol:  executed at node i when
//                            a event is received from node j

case event type of

data event m:
     if m is received over shortcut and
        MOMPChannelRec(M).shutdownEventID[ r ] = m.id
          send clear-shortcut to
                            MOMPChannelRec(M).remoteShortcutsFrom[ r ];
          remove entry r from lists;
     else
          if m is not a duplicate
               deliver m to local members;
               send copies of m to all nodes in
                                 MOMPChannelRec(M).ShortcutList except j;
          end if
     end if

shortcut-added(M):
     join channel M;
     MOMPChannelRec(M).remoteShortcuts[ r ] = j;
     MOMPChannelRec(M).shutdownEventID[ r ] = next EventID on channel M;

clear-shortcut(M):
     remove j from MOMPChannelRec(M).ShortcutList;
     delete from channel M;

end case
```

Nearest Insertion

Der RB-Algorithmus verwendet nur den zugrundeliegenden statischen Baum für das Versenden von Events. Dies kann zu Performanceproblemen in der Nähe der Wurzel führen. In gut verbundenen Netzwerken wird darüberhinaus die vorhandene Konnektivität überhaupt nicht verwendet. Der NI-Algorithmus nutzt nun die Konnektivität des Netzwerks aus, um die Performance der Multicasts zu verbessern und die Last besser zu verteilen. Zu beachten ist, daß der NI-Algorithmus nur auf der Ebene eines Subnetzes (zum Beispiel eines LANs) ausgeführt wird. In jedem Subnetz ist ein Koordinator vorhanden.

Der Unterschied zum RB-Algorithmus liegt darin, daß wenn der Baum erweitert werden muß, nun alle Links eines Knotens potentielle Kandidaten für die Erweiterung sind. Im folgenden betrachten wir die möglichen Aktionen.

- **Gruppe beitreten**
  Wenn der beitretende Knoten *j* die Gruppe bereits kennt, der GroupRec also bereits existiert, setzt er nur noch das Memberflag im GroupRec. Andernfalls sendet er *join-request* zu seinem Koordinator. Der Koordinator sendet den *join-request* nun wieder in das Subnetz, und bearbeitet ihn nach dem RB-Algorithmus. Als Antwort erhält er vom RB-Algorithmus die Meldung *participant*, falls die Gruppe in einem anderen Subnetz bekannt ist. Aus seinem eigenen Subnetz erhält er von jedem Gruppenteilnehmer eine Abschätzung, wie weit dieser von *j* entfernt ist. Der am nächsten liegende Teilnehmer der Gruppe wird vom Koordinator ausgewählt und an *j* gesendet. Dann wird der Gruppenbaum erweitert, indem die Knoten und Links auf dem kürzesten Weg von *j* zum nächsten Gruppenteilnehmer markiert und in den Baum aufgenommen werden.

- **Gruppe verlassen**
  Arbeitet fast genauso wie beim RB-Algorithmus. Der Ast des Gruppenbaums wird solange abgeschnitten bis entweder ein Member erreicht ist, oder ein Knoten zwei Nachbarn hat. Wenn der Knoten, der die Gruppe verläßt, keine Nachbarn mehr hat, wird der Koordinator benachrichtigt.

- **Event senden**
  Arbeitet wie beim RB-Algorithmus, nur daß der Event zum Koordinator gesendet wird, wenn der sendende Knoten die Gruppe nicht kennt. Der Koordinator sorgt dann für die Weiterleitung.

In einem LAN mit Broadcastkommunikation (wie dem Ethernet) ist das Aufbauen eines Gruppenbaums für den NI-Algorithmus nicht nötig, da jeder Knoten direkt erreicht werden kann. Es genügt dann, wenn die Koordinatoren pro Gruppe einen Zähler verwalten wieviele Hosts der Gruppe im LAN vorhanden sind und die Koordinatoren über das Hinzukommen und Verlassen von Hosts in den Gruppen informiert werden.

Der folgende Code wird auf Knoten *i* ausgeführt.

```
// Nearest Insertion:   executed at node i

case local operation of

join group M:
     if GroupRec(M) exists then
          GroupRec(M).status := member;
     else
          allocate GroupRec(M);
          GroupRec(M).status := member;
          send join-request to the coordinator;
          let l be the nearest participant received from coordinator,
                    and (i,j,m,n, ...,l) be the shortest path;
          add j to GroupRec(M).NeighbourList;
          send extend-tree with path (m,n,...,l) to j;
     end if

delete from group M:
     if Size(GroupRec(M).NeighbourList) > 1 then
          GroupRec(M).status := non-member;
     else
          if Size(GroupRec(M).NeighbourList) = 1 then
               send delete-request to node in GroupRec(M).NeighbourList;
          else
               send delete-request to coordinator;
          end if
          deallocate GroupRec(M);
     end if

send event m to group M:
     if GroupRec(M) exists then
          deliver m to local members;
          send a copy of m to each node in GroupRec(M).NeighbourList;
     else
          search a participant;
          if a participant was found then
               // M is known in subnet
               send m to participant;
          else
               // M is unknown in subnet
               send m to coordinator;
          end if
     end if

end case
```

Dieser Code wird bei Empfang einer Nachricht von Knoten *j* auf Knoten *i* ausgeführt.

```
// Nearest Insertion:   executed at node i when
//                      a event is received from node j

case event type of

data event m:
     // same as in Restricted Broadcast
     deliver m to local members;
     send copies of m to all nodes in GroupRec(M).NeighbourList except j;

delete-request:
     // same as in Restricted Broadcast
     remove j from GroupRec(M).NeighbourList;
     if GroupRec(M).status = non-member and
          Size(GroupRec(M).NeighbourList) = 1 then
          send delete-request to node in GroupRec(M).NeighbourList;
          deallocate GroupRec(M);
     end if

join-request:
     send join-request to nodes in GroupRec(M).NeighbourList except j;
     find the shortest path to node received in event;
     report the path to the coordinator;

extend-request:
     let (m,n,...,l) be the path received in request;
     if GroupRec(M) exists then
          add j to GroupRec(M).NeighbourList;
     else
          allocate GroupRec(M);
          GroupRec(M).status := non-member;
          add j and m to GroupRec(M).NeighbourList;
          send extend-request to m with (n,...,l) as path;
     end if

end case
```

C. Literatur

In den Literaturangaben sind Verweise auf "Requests for Comments" (RFC) des Internets enthalten. RFCs sind unter anderem auf dem FTP-Server des Rechenzentrums der Universität Stuttgart verfügbar (ftp://ftp.rus.uni-stuttgart.de/pub/doc/standards/rfc).

[Agha87] Agha, Gul A.:
ACTORS: a model of concurrent computation in distributed systems
MIT Press, Cambridge, MA, 1987

[Ahamad90] Ahamad, Mustaque:
Multicast communication in distributed systems
IEEE Computer Society Press Technology Series, 1990

[Baumann97a] Baumann, Joachim; Hohl, Fritz; Radouniklis, Nikolaos; Straßer, Markus; Rothermel, Kurt:
Communication Concepts for Mobile Agent Systems
IPVR, Universität Stuttgart, 1997

[Baumann97b] Baumann, Joachim; Beck, Bernhard; Radouniklis, Nikolaos; Szasz, Victor; Zepf, Matthias:
Gespräche und Protokolle der Eventgruppe
IPVR, Universität Stuttgart, 1996/97 (unveröffentlicht)

[Beck96] Beck, Bernhard:
Konzeption und Implementierung eines graphischen Monitors für ein Mobile-Agenten-System
Studienarbeit Nr. 1523
IPVR, Universität Stuttgart, 1996

[Belkeir89] Belkeir, Naser E.; Ahamad, Mustaque:
Low Cost Algorithms for Message Delivery in Dynamic Multicast Groups
Procedings of IEEE Ninth International Conference on Distributed Computing, June 1989, S. 110-117
auch in: [Ahamad90] S.73 - 80

[Birman87] Birman, Kenneth P.; Joseph, Thomas A.:
Reliable Communication in the Presence of Failures
ACM Transactions on Computer Systems, (5), 1, February 1987

[Chang84] Chang, Jo-Mei; Maxemchuck, N. F.:
Reliable Broadcast Protocols
ACM Transactions of Computer Systems, (2), 3, August 1984

[Deering88] Deering, Stephen E.:
Multicast Routing in InterNetworks and Extended LANs
ACM Computer Communications Review (18), 4, 1988, S.55-64
auch in: [Ahamad90], S. 63-72

[Deering89] Deering, Stephen E.:
Host Extensions for IP Multicasting
RFC 1112, August 1989

[Gallager83] Gallager, R.G.; Humblet, P.A.; Spira, P.M.:
A Distributed Algorithm for Minimum-Weight Spanning Trees
ACM Transactions on Programming Languages and Systems, (5), 1, 1983, S. 66-77

[Harrison95] Harrison, Colin G.; Chess, David M.; Kershenbaum, Aaron:
Mobile Agents: Are they a good idea?
IBM Research Division, T. J. Watson Research Center
Yorktown Heights, NY 10598, 1995

[Hohl95] Hohl, Fritz:
Konzeption eines einfachen Agentensystems und Implementation eines Prototyps
Diplomarbeit Nr. 1267
IPVR, Universität Stuttgart, 1995

[Jalote94] Jalote, Pankaj:
Fault Tolerance in Distributed Systems
PTR Prentice Hall, Eaglewood, N.J. 07632, 1994

[Kaashoek91] Kaashoek, M.F.; Tanenbaum, Andrew S.:
Group Communication in the Amoeba Distributed Operating System
Proceedings of the
11th Conference on Distributed Computing Systems, 1991

[Kuhn97] Kuhn, Wilfried:
Ressourcenkontrolle in einem Mobile Agenten System
Diplomarbeit Nr. 1468
IPVR, Universität Stuttgart, 1997

[Lamport78] Lamport, Leslie:
Time, Clocks, and the Ordering of Events in a Distributed System
Communications of the ACM (21), 7, July 1978, S. 558 - 565

[Levine97] Levine, Brian Neil; Garcia-Luna-Aceves, JJ:
A Comparison of Known Classes of Reliable Multicast Protocols
Computer Engineering Department, University of California,
Santa Cruz, CA, 1997 (unveröffentlicht)

[Mills92] Mills, D.:
Network Time Protocol (v3)
RFC 1305, 09.04.1992

[Mogul84] Mogul, J.:
Broadcasting Internet datagrams (in the presence of subnets)
RFC 919 und RFC 922, 01.10.1984

[MOLE97] **MOLE-Projektseite** im WWW
http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole

[MQseries97] IBM Laboratories Hursley Park:
IBM MQseries Homepage
http://www.hursley.ibm.com/mqseries

[OMG97] Object Management Group (OMG):
OMG Homepage
http://www.omg.com/

[Orbix96] IONA Technologies Ltd.:
OrbixTalk™ White Paper
IONA Technologies Ltd, Dublin, April 1996
von: ftp://ftp.iona.com/www/Orbix/Talk/WhitePaper/OrbixTalk.A4.ps.gz

[Panzieri88] Panzieri, Fabio; Shrivastava, Santosh K.:
Rajdoot: A Remote Procedure Call Mechanism supporting Orphan Detection and Killing
IEEE Transactions on Software Engineering, (14), 1, January 1988

[Raynal88] Raynal, Michel:
Distributed algorithms and protocols
Chichester [u.a.]: Wiley, 1988

[Rieg96] Rieg, Berthold:
Robuste Warteschlangen für Nachrichten als Kommunikationsmechanismus in einem Mobile-Agenten-System: Konzeption und Implementierung
Studienarbeit Nr. 1528
IPVR, Universität Stuttgart, 1996

[Rieg97] Rieg, Berthold:
Messaging and Queueing
IPVR, Universität Stuttgart, 1997

[Shapiro92] Shapiro, Marc; Dickman, Peter; Plainfossé, David:
SSP Chains: Robust, Distributed References supporting acyclic garbage collection
Rapport de Recherche No. 1799
INRIA, Rocquencourt, Frankreich, November 1992
von: http://www.inria.fr/

[Szasz97] Szasz, Victor:
Protokolle für Synchronisation in einem System mobiler Software-Agenten
Studienarbeit Nr. ????
IPVR, Universität Stuttgart, 1997

[TIBCO96] TIBCO Inc.:
Rendezvous Documentation Set: TIBCO Rendezvous Information Bus
Release 2.0, July 1996
TIBCO Inc. Palo Alto, CA, 1996

[Villinger96] Villinger, Klaus:
Einkaufs- und Verkaufsagenten für den elektronischen Markt
Diplomarbeit Nr. 1403
IPVR, Universität Stuttgart, 1996

[Waitzman88] Waitzman, D; Partridge, C.; Deering, S.:
Distance Vector Multicast Routing Protocol
RFC 1075, November 1988

[Wall80] Wall, David W.:
PhD Thesis: Mechanisms for Broadcast and Selective Broadcast
Technical Report No. 190
Computer Systems Laboratory
Stanford University, June 1980
Kurzfassung in [Ahamad90] S. 1 - 9

[Webster92] Spencer, David (compil.):
Webster's new world dictionary of computer terms
4. ed. - New York : Prentice Hall, 1992

[White94] White, James E.:
Telescript Technology:
The Foundation for the Electronic Marketplace
White Paper, General Magic Inc., Sunnyvale, CA, 1994

[Zepf96] Zepf, Matthias:
Konzeption und Implementierung eines einfachen Orphan-Detection-Mechanismus für ein Mobile-Agenten-System
Studienarbeit Nr. 1552
IPVR, Universität Stuttgart, 1996

D. Glossar

Agent

Ein Agent ist ein Objekt, welches einen Ausführungspfad (Thread) besitzt und in einem Agentensystem in Auftrag und zu Nutzen eines anderen Agenten oder eines Menschen autonom arbeiten kann. In MOLE müssen Agenten von der Klasse mole.Agent abgeleitet sein (siehe auch Kapitel I.3.1). Mobile Agenten haben zusätzlich die Möglichkeit zwischen Locations zu migrieren (siehe Kapitel 1.3.2).

Agentengruppe

Eine (un-)geordnete Menge von Agenten, die in einer Beziehung zueinander stehen. Eine Agentengruppe besitzt einen privaten Eventchannel, sowie einen internal und einen external Group Coordinator (siehe auch Kapitel I.3.5).

Agentensystem

Ein Agentensystem stellt eine Laufzeitumgebung für Agenten zur Verfügung, die über besondere Sicherheitsmechanismen verfügt, damit die Agenten keinen direkten Zugriff auf Ressourcen des Rechners haben. Das dieser Diplomarbeit zugrundeliegende Agentensystem heißt MOLE und wird am IPVR der Universität Stuttgart entwickelt (siehe auch Kapitel I.3.3).

Broadcast

Das gleichzeitige Versenden einer Nachricht an alle Knoten eines Netzwerks.

Consumer

Ein Objekt (meistens ein Agent), das Events von einem Eventchannel empfangen kann.

CORBA, Common Object Request Broker Architecture

Eine standardisierte Architektur für netzwerktransparente Kommunikation von Applikations- und Systemobjekten in verteilten Systemen aus heterogenen Susbsystemen. Zur Realisierung werden ORBs verwendet (siehe Kapitel III.1).

Engine

Repräsentiert das Agentensystem im Betriebssystem und stellt die Laufzeitumgebung des Agentensystems zur Verfügung. In einem Betriebssystemprozeß läuft genau eine Engine, die eventuell mehrere Locations verwaltet (siehe Kapitel I.3.4)

Event

Die Nachricht, daß ein bestimmtes Ereignis bei einem Supplier eingetreten ist oder bei einem Consumer ausgelöst werden soll. Events werden über Eventchannels gesendet, die vom Eventmanager verwaltet werden.

Eventchannel

Ein logischer Kommunikationskanal zwischen Suppliern und Consumern, über den Events versendet werden und der gewisse Dienstmerkmale garantiert.

Eventmanager
Die Gesamtheit aller Prozesse und Kommunikationspfade, die für die Verwaltung und das Versenden von Events nötig sind. Der Eventmanager stellt die durch Eventchannels definierte Funktionalität tatsächlich zur Verfügung.

GC, Group Coordinator, internal / external
Notwendiges Bestandteil einer Agentengruppe. Die Gruppenbedingung wird im internal Group Coordinator verwaltet. Über den external Group Coordinator kann die Gruppe Mitglied in anderen Gruppen werden (siehe Kapitel I.3.5.).

Hop
In paketvermittelten Netzwerken die Übertragung einer Nachricht von einem Rechner zum nächsten Rechner auf dem Weg zum Ziel.

Hostgruppe
Bei IP Multicast die kleinste addressierbare Einheit. Ein Rechner kann Mitglied in einer Hostgruppe werden, indem er IGMP verwendet (siehe Kapitel III.2.1).

IIOP, Internet Inter-ORB Protocol
Standardisiertes Protokoll für die Kommunikation zwischen mehreren ORBs, zum Austausch von CORBA-Objekten und Verwaltungsinformationen.

IP, Internet Protocol
IP ist ein verbindungsloser, datagrammorientierter Netzwerk-Dienst zum Versenden von Datenpaketen zwischen Rechnern. Zur Zeit aktuell ist Version 4 (IPv4), die im Laufe der nächsten Jahre durch Version 6 (IPv6 oder IPng, "next generation") ersetzt werden soll.

IP Multicast
Multicastservice, der im heutigen Internet überwiegend eingesetzt wird (siehe Kapitel III.2.1).

IGMP, Internet Group Multicast Protocol
Informationsprotokoll für Multicastrouter, um Informationen über die in einem lokalen Netz vorhandenen Hostgruppen zu erhalten (siehe Kapitel III.2.1).

Knotengruppe
In unserem Gruppenmodell die Menge von Knoten, die Teilnehmer desselben Eventchannels verwalten oder Events für diesen Eventchannel weiterleiten.

Location
Aufenthaltsort von Agenten. Agenten können sich nur in Locations aufhalten und zwischen ihnen migrieren. Die zugrundeliegende Infrastruktur ist für sie nicht sichtbar.

MOLE
Agentensystem, daß am IPVR der Universität Stuttgart entwickelt wird.

MSB, minimal spannender Baum über Knotenmenge V und Kantenmenge E

Ein Graph G mit folgenden Eigenschaften: enthält keine Zyklen, alle Knoten aus V sind in G enthalten, alle Kanten in G sind in E enthalten, die Summe der Kantengewichte von G ist minimal, G ist zusammenhängend.

Multicast

Das gleichzeitige Versenden einer Nachricht an eine Gruppe von Knoten des Netzwerks.

Multicastrouter (IGMP, IP Multicast)

Ein Router, der Multicastnachrichten effizient weiterleiten kann (siehe Kapitel III.2.1)

Netzpartitionierung

Der Alptraum im Bereich der verteilten Systeme. Ein großes Netzwerk zerfällt durch Ausfall eines oder mehrer Rechner in getrennte Teilnetze, die nicht miteinander kommunizieren können.

OMG, Object Management Group

Internationale Standardisierungsgruppe mit dem Ziel netzwerktransparente Kommunikation heterogener Systeme zu ermöglichen. Zuständig für die (Weiter-)Entwicklung von CORBA (siehe Kapitel III.1).

ORB, Object Request Broker

Ein Programm, das die durch CORBA definierte Architektur implementiert und den Anwender bei der Nutzung unterstützt (siehe Kapitel III.1).

PDA, personal digital assistant

Ein kleiner, tragbarer "Westentaschen"-Computer mit verhältnismäßig geringer Rechenleistung. In letzter Zeit zunehmend mit Funktelefonen kombinierbar. Beispiele sind Apples Newton, Sharps Organizer oder PC-Sub-Notebooks.

Piggybacking

Speichern von Kontrollinformationen eines Protokolls in normalen Datenpaketen. Wird gerne zur Verringerung der benötigten Bandbreite von Protokollen verwendet.

Remote Execution / Evaluation

Entfernte Ausführung von Programmen auf einem anderen Rechner. Konzept ähnlich zu Agenten mit dem Unterschied, daß Agenten autonom zu anderen Rechnern weitermigrieren können.

RPM, Reverse Path Multicasting

Heute im Internet verwendeter Basisalgorithmus für Multicastkommunikation (siehe Kapitel III.2.1).

SPOF, Single Point Of Failure

Eine (meist zentrale) Einheit eines verteilten Systems bei deren Ausfall die Funktions-

fähigkeit des gesamten Systems nicht mehr gewährleistet ist. SPOF sollen in verteilten Systemen möglichst vermieden werden.

Supplier
Ein Objekt (meistens ein Agent), das Events in einen Eventchannel senden kann.

TCP, Transmission Control Protocol
Ein Protokoll, das auf IP aufsetzt und eine gesicherte Datenübertragung zwischen zwei Endsystemen gewährleistet. TCP überträgt einen seriellen, unstrukturierten Bytestrom, der zur Übertragung in einzelne Segmente aufgeteilt wird (Pufferung, erst wenn ein Segment voll ist, wird es gesendet). Über eine TCP-Verbindung können gleichzeitig Daten in beide Richtungen gesendet werden.

TTL, time-to-live
Zähler im Header eines IP-Pakets, der anzeigt wie oft dieses Paket noch weitergeleitet werden darf. Erreicht TTL den Wert 0 so wird das Paket nicht mehr weitergeleitet. TTL wurde ursprünglich in Sekunden angegeben, inzwischen ist es üblicher die Anzahl der verbleibenden Hops anzugeben.

UDP, User Datagramm Protocol
Sehr einfaches paketorientiertes Protokoll, das auf IP aufsetzt. Reihenfolgefehler und Duplikate sind möglich.

Unicast
Das Senden einer Nachricht an genau einen Empfänger.

Zum Schluß ...

Ich versichere, daß ich diese Arbeit selbstständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.

Stuttgart, den 14. Mai 1997