Universität Karlsruhe
Fakultät für Informatik
Institut für Telematik
76128 Karlsruhe

Architektur vernetzter Systeme

Seminar Sommersemester 1999

Herausgeber:
Dr. Arnd Grosse
Rainer Ruggaber
Dr. Jochen Seitz

Universität Karlsruhe
Institut für Telematik

Interner Bericht 13/99
ISSN 1432–7864

Zusammenfassung

Der vorliegende interne Bericht enthält die Beiträge von Studenten zum Seminar „Architektur vernetzter Systeme", das im Sommersemester 1999 am Institut für Telematik der Universität Karlsruhe (TH) stattgefunden hat. Darin enthaltene Themengebiete sind Erweiterungen von CORBA, die Unterstützung mobiler Anwendungen sowie Verfahren zur Optimierung großer verteilter Systeme.

Abstract

This technical report comprises student papers within assignments for the seminar "Architektur vernetzter Systeme". It took place at the Institute of Telematics of the University of Karlsruhe in summer 1999. Main topics for discussion contained CORBA extensions, tools and systems for mobile applications and also optimization concepts for large distributed systems.

Inhaltsverzeichnis

Vorwort

Neben der rein technischen Weiterentwicklung der Kommunikationsinfrastruktur erfolgt parallel eine fortschreitende Änderung auch im Middleware- und Anwendungsbereich durch neue Einsatzgebiete und Paradigmen für vernetzte Systeme. Der hier vorliegende Seminarband widmet sich dieser Thematik und entstammt der Reihe „Architektur vernetzter Systeme". Er betrachtet konkret neue Ansätze und Systeme zur Realisierung und Optimierung verteilter Systeme und Anwendungen.

Der Seminarband läßt sich in drei Teilgebiete untergliedern. Zunächst erfolgt eine differenziertere Betrachtung der für verteilte Systeme neben DCOM relevanten CORBA-Architektur durch folgende drei Seminarthemen:

Programmierung des CORBA POAs (Portable Object Adapter)
Ein Object Adapter sorgt in CORBA auf Server-Seite für die Zuordnung und/oder Erzeugung von diensterbringenden (Server-)Objekten. Das vorliegende Seminarthema stellt den neuen Portable Object Adapter in CORBA vor, der gegenüber dem bis dahin verwendeten Basic Object Adapter sowohl größere Flexibilität als auch Funktionalität besitzt.

Echtzeit-CORBA
Mit CORBA steht eine leistungsfähige Middleware-Architektur zu Verfügung. Zentraler Bestandteil dieser Architektur ist der Object Request Broker (ORB), der für die Kommunikation zwischen einzelnen Objekten sorgt. Im Hinblick auf Echtzeitanwendungen erfüllen existierende CORBA-Implementierungen nur unzureichend geltende Dienstgüteanforderungen. Das vorliegende Seminarthema diskutiert zunächst die Nachteile von CORBA gegenüber Anforderungen von Echtzeitanwendungen und stellt danach mit TAO eine echtzeitfähige CORBA-Implementierung vor.

A/V Streaming in CORBA
CORBA wurde zunächst zur Realisierung dienstnehmer-/dienstgeberbasierter verteilter Anwendungen entworfen. Eine effiziente Unterstützung kontinuierlicher Datenströme, wie sie gerade im Audio- und Videoumfeld vorzufinden sind, war dabei nicht vorgesehen. Erweiterungen von CORBA in diese Richtung stellt die vorliegende Seminararbeit vor.

Ein zweites Teilgebiet des Seminarbands umfasst die Präsentation von ausgewählten Ansätzen, die Lösungen für die Realisierung verteilter als auch zugleich mobiler Anwendungen bieten:

Jini – Java Intelligent Network Infrastructure
Neben den Vorteilen von Java als Sprache für eine vereinfachte und damit weniger fehlerträchtige Programmierung verteilter Anwendungen liegt eine weitere Stärke in der umfangreichen Unterstützung durch Bibliotheken und neuer Programmierkonzepte. Eines dieser Konzepte ist Jini, das im Wesentlichen gegenüber dem Trading in CORBA eine vereinfachte Vermittlungsfunktionalität zwischen Dienstnehmer und Dienstgeber bereitstellt.

MOBIWARE – Eine Plattform für verteilte Anwendungen mobiler Teilnehmer
MOBIWARE ist eine Middleware-Plattform und behandelt das Problem der Integration mobiler Teilnehmer in ein ATM-Netz. Hierzu wird ein umfassendes Software-Paket zur Unterstützung bereitgestellt. Primärziel ist dabei die Beibehaltung von Qualitätsanforderungen einzelner Anwendungen trotz zugrunde liegender Mobilität.

Entwurf verteilter mobiler Anwendungen mit ROVER
Kleine mobile Endgeräte besitzen gegenüber fest installierten Geräten deutliche Nachteile im Hinblick auf ihre Ressourcenausstattung wie Akkus und Hardware. Dieses bedingt bei der Realisierung von auf mobilen Endgeräten platzierten Anwendungen die Berücksichtigung sowohl ressourcenschonender Mechanismen als auch Kommunikationsprotokollen. Mit ROVER steht eine Entwicklungsumgebung für diese Problematik zur Verfügung.

Vernetzung über GSM – Die Mowgli-Architektur
Das Mowgli-Projekt unterstützt spezifisch die Anbindung mobiler Rechner über GSM an ein Festnetz. Zielsetzung ist es – trotz GSM-Kommunikation (derzeit 9,6 kbit/s) – leistungsfähige Anwendungen auf den Mobilrechnern betreiben zu können.

Mole – Ein System für mobile Agenten
Ein mobiler Agent ist Bestandteil eines neuen Programmierparadigmas zur Realisierung verteilter Anwendungen, bei dem nicht mehr entfernte Anfragen gestellt werden, sondern ein Stück Code zu potentiell geeigneten Orten migrieren kann und dort ausgeführt wird. Dieses erfordert geeignete Verfahren zur Laufzeitunterstützung, welche exemplarisch am System Mole vorgestellt werden.

Mit der neu hinzukommenden Mobilität von Teilnehmern und dem Anwachsen der Teilnehmerzahlen in Systemen ist eng das Problem der Optimierung der Leistungsfähigkeit und die Skalierbarkeit verbunden:

Objektlokation für mobile Anwendungen
Bestehende Namenverwaltungssysteme sind hierarchisch strukturiert und setzen nur eine geringe Änderungswahrscheinlichkeit eingetragener Namen und Adressen voraus. Diese Annahme ist jedoch im Umfeld hochgradig mobiler Anwendungen nicht mehr tragbar und skaliert nicht. Ein hierfür besser geeigneter Suchdienst wird in der vorliegenden Seminararbeit mit dem „Globe Location Service" diskutiert.

Neue Ansätze zur (weiträumigen) Verteilung in vernetzten Systemen
Die Informationssuche innerhalb des WWW ist durch eine dienstnehmerzentrierte Vorgehensweise geprägt. Dies führt zu einer unzureichenden Ausnutzung vorhandener Ressourcen und zur partiellen Überlastung einzelner Server und Kommunikationsverbindungen. Neue Ansätze, die bessere Ressourcennutzung versprechen, werden in der genannten Seminararbeit diskutiert.

Dynamisch verteilte Objekte
Flexibilität und Wartbarkeit sind unternehmenskritische Faktoren, die insbesondere bei der Realisierung verteilter Anwendungen von großer Bedeutung sind. Sowohl durch die Verteilung von Objekten als auch durch deren anwendungsspezifische Zerlegung und damit gezieltere Verwendung existieren Mechanismen, die eine Verbesserung der Problematik ermöglichen und in der Seminararbeit diskutiert werden.

Bevor nun die einzelnen Ausarbeitungen der Seminarbeiträge präsentiert werden, möchten wir allen beteiligten Studenten für ihre engagierte Mitarbeit danken, ohne die weder der Erfolg des Seminars noch die Anfertigung des vorliegenden Berichts möglich gewesen wäre. Hierzu haben auch die nach den einzelnen Vorträgen stattfindenden Diskussionen maßgeblich beigetragen.

Karlsruhe, im Oktober 1999

Arnd Grosse Rainer Ruggaber Jochen Seitz

Programmierung des CORBA POAs (Portable Object Adapter)

Jörg Fröhlich

Kurzfassung

In diesem Papier werde ich Inhalte betrachten, die im Zusammenhang mit CORBA Object Adaptern stehen. Nach einer kurzen Einführung in CORBA werde ich mich darauf konzentrieren, was ein Object Adapter ist, und was seine Rolle in einem CORBA-basierten System ausmacht. Ich werde auf die CORBA Object Adapter: Basic Object Adapter (BOA) und Portable Object Adapter (POA) eingehen wobei ich schwerpunktmäßig auf Probleme des BOAs eingehen möchte, die im POA gelöst werden. Auch Beispiele für Umsetzungsmöglichkeiten in realen CORBA C++ Applikationen unter Verwendung des POAs sollen gezeigt werden.

1 Einleitung

2 Die CORBA Architektur

Bei der Entwicklung der Common Object Request Broker Architecture (CORBA) kam zwei Ideen besondere Bedeutung zu: erstens, daß Objekttechnologie in komponentenbasierte Systeme mündet und zweitens, daß verteilte Systeme eher die Regel als die Ausnahme sein werden [SaCu98].

Ziel von Distributed Object Systems [SaCu98] ist es Objekten auf verschiedenen Computern ein Zusammenarbeiten zu ermöglichen, als wenn sie sich auf dem selben Rechner befinden würden. [PySc98]ergänzen: CORBA ist ein Standard für Distributed Object Computing (DOC) Middleware. DOC Middleware ist angesiedelt zwischen Client und Server. Sie vereinfacht die Entwicklung von Applikationen durch die Bereitstellung einer einheitlichen Sicht der heterogenen Netzwerke und BS Schichten.

Clients können Applikationen starten ohne sich um folgende Punkte kümmern zu müssen:

- Lokalisierung von Objekten,
- Programmiersprache (unterstützt werden u.a. C, C++, Java, Ada95, COBOL und Smalltalk),
- OS Plattform (CORBA ist u.a. unter Win32, UNIX, MVS, VxWorks, Chorus und LynxOS),
- Kommunikationsprotokolle und Interconnects (inkl.: TCP/IP, IPX/SPX, FDDI, ATM, Ethernet)
- Hardware (CORBA schirmt Anwendungen von Unterschieden in der Hardware ab)

Im Zentrum von DOC middleware stehen Object Request Brokers (ORBs), definiert durch Standards, wie zum Beispiel CORBA, DCOM und Java RMI. Averkamp und Co. [APRA98], geben an, daß bereits die Auswahl zwischen den möglichen Standards, bei der Wahl des geeigneten Werkzeugs für die komponentenbasierte Entwicklung, einen großen Stolperstein darstellt. ORBs eliminieren viele Fehlerquellen und Aspekte die sonst einer Portabilität bei der Entwicklung und Wartung von verteilten Systemen im Wege stünden. Dies geschieht durch Automatisierung von Standardprogrammieraufgaben, wie z.B. Objektlokalisierung, Objektaktivierung, Parameter Marshaling, Fault Recovery und Sicherheit.

Ein Object Adapter ist integraler Bestandteil von CORBA und unterstützt einen ORB, indem er Anfragen von Clients an Server Objekt Implementationen (Diensterbringer) weiterleitet. Von einem Object Adapter bereitgestellte Dienste beinhalten dabei: (1) Generierung und Interpretation von Objektreferenzen, (2) Aktivierung und Deaktivierung von Diensterbringer, (3) Demultiplexing von Requests um Objektreferenzen auf die zugehörigen Diensterbringer zu übertragen. (4) Zusammenarbeit mit dem Interface Definition Language (IDL) Skeleton um Operationen auf Diensterbringer anwenden zu können [PySc98].

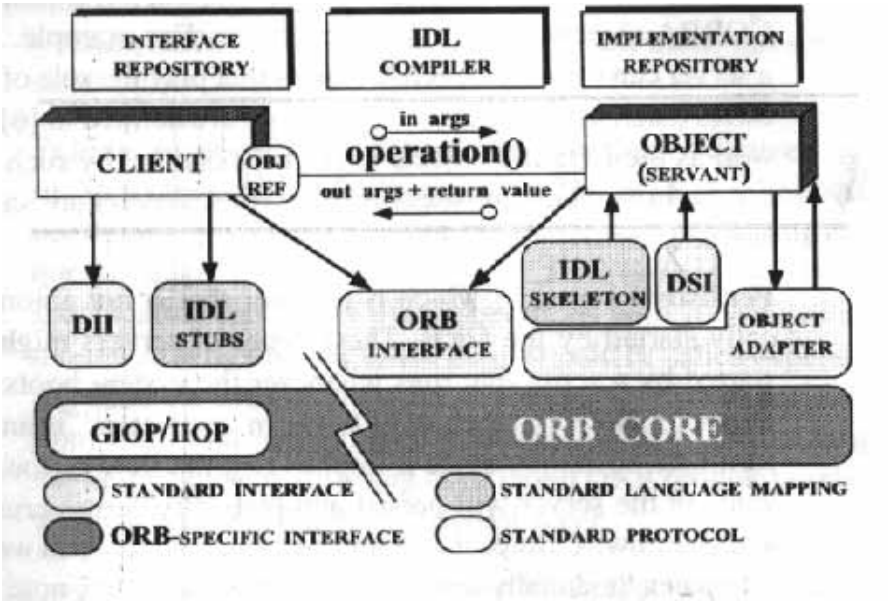


Abbildung 1: Components in the CORBA Reference Model [PySc98]

Die Komponenten im CORBA Referenzmodell:

Diensterbringer: Setzt die Operationen, so wie sie von einem OMG Interface Definion Language (IDL) Interface definiert werden, um. Diensterbringer werden normalerweise objektorientiert implementiert. Alternativ können in nicht objektorientierten Programmiersprachen hierfür typischerweise Funktionen und Structs verwendet werden.

Client: führt Applikationen durch, indem er die Objektreferenzen, die auf die Diensterbringer verweisen erwirbt und Operationen auf diesen Diensterbringern auslöst. Diensterbringer müssen sich nicht auf dem gleichen System befinden wie Clients. Idealer Weise sollte es genauso einfach sein auf einen Diensterbringer auf einem anderen System zuzugreifen, wie auf einen, der sich auf dem eigenen befindet, d.h. object->operation(args).

ORB Core: Wenn ein Client eine Operation durch einen Diensterbringer ausführen läßt, ist der ORB Core dafür zuständig die Anfragen dem Diensterbringer zu liefern und gegebenenfalls dem Client die Antworten zu überbringen. Für Diensterbringer, welche auf einem anderen System angesiedelt sind kommuniziert ein CORBA-compliant ORB

Core via Internet Inter-ORB Protocol (IIOP), das eine Spezialform des General Inter-ORB Protocols (GIOP) darstellt, welche auf das TCP Transport Protokoll aufsetzt. Ein ORB Core ist typischerweise in Form einer, mit Client und Server verknüpften, run-time Bibliothek implementiert.

ORB Interface: Ein ORB ist eine logische Einheit, die auf unterschiedliche Art implementiert werden kann, z.B. durch ein oder mehrere Prozesse oder eine Reihe von Bibliotheken. Um Anwendungen von den Details ihrer Implementation logisch zu trennen definiert die CORBA Spezifikation ein abstraktes Interface für einen ORB. Dieses ORB Interface stellt Standardoperationen zur Verfügung, die 1. den ORB initialisieren und seine Ausführung abschließen, 2. Objektereferenzen in Strings konvertieren und umgekehrt und 3. Argumentenlisten für Anfragen generieren, welche von Dynamic Invokation Interfaces (DII) stammen.

OMG IDL Stubs und Skeletons: dienen der Verknüpfung von Client und Diensterbringer und dem ORB. Stubs stellen ein statisches Invocation Interface (SII) zur Verfügung, welches Anwendungsdaten in ein anwendungs- und systemunabhängiges Übertragungsformat umwandelt. Umgekehrt wandeln Skeletons dieses Übertragungsformat wieder in das, der Anwendung entsprechende Format um.

IDL Compiler: transformieren OMG IDL Definitionen in Programmiersprachen der Applikationen, wie z.B. Java oder C++. Um Sprachtransparenz zur Verfügung stellen zu können, eliminieren IDL Compiler gewöhnliche Fehler in der Programmierung von Netzwerken und bieten die Möglichkeit automatische Optimierung bei der Compilierung vornehmen zu lassen.

Dynamic Invocation Interface (DII): erlaubt Clients Anfragen während der Laufzeit zu generieren. Das DII ermöglicht Clients außerdem Defered Synchronous Calls zu tätigen, welche dazu dienen, die Anfrage- und Antwortportionen der in beide Richtungen verlaufenden Operationen unabhängig zu machen. Hierdurch wird verhindert, daß ein Client blockiert ist, bis der Diensterbringer geantwortet hat.

Dynamic Skeleton Interface (DSI): ist das Pendant für den Server zum DII des Clients. Das DSI ermöglicht es einem ORB Anfragen an einen Diensterbringer zu übermitteln, dem zum Zeitpunkt der Compilierung keine Informationen über das IDL Interface, welches es implementiert, vorliegen. Anfragende Clients benötigen keine Kenntnisse darüber, ob ein Server ORB ein statisches oder ein dynamisches Skeleton verwendet. Ebenso müssen Server nicht wissen, ob ein Client ein DII oder ein SII für seine Anfragen verwendet.

Objekt Adapter: verknüpft einen Diensterbringer mit einem ORB, ist für das Demultiplexing von eingehenden Anfragen zuständig und ruft die passende Operation auf diesem Diensterbringer auf. Jüngste Verbesserungen der Portabilität innerhalb von CORBA definieren den Portable Object Adapter (POA), welcher mehrere Nested-POAs pro ORB zuläßt. Object Adapter ermöglichen es einem ORB verschiedene Typen von Diensterbringern, welche ähnliche Anfragen haben, zu unterstützen. Diese Architektur hat einen kleinen und einfachen ORB zum Ergebnis, welcher eine große Bandbreite von Objekten unterstützt (bezogen auf Granularität, Lebensdauer, Policies, Stil der Implementierung und anderer Eigenschaften).

Interface Repository: stellt Laufzeit-Informationen über IDL Interfaces zur Verfügung. Mit dieser Information ist es einem Programm möglich mit einem Objekt umzugehen, dessen Interface zum Zeitpunkt der Compilierung des Programmes nicht bekannt war. Nun ist es möglich festzustellen, welche Operationen auf dieses Objekt angewendet werden

dürfen und auf es zuzugreifen. Des weiteren stellt das Interface Repository einen Speicherplatz zur Verfügung indem ergänzende Informationen, welche im Zusammenhang mit Interfaces ORB Objekten stehen, abgelegt werden können. (Z. B.: Stub/Skeleton Type Bibliotheken)

Implementation Repositories: enthält Informationen, welche es dem ORB erlauben Diensterbringer zu lokalisieren und zu aktivieren

GIOP/IIOP: Ein besonderes Merkmal des CORBA-Standards ist laut Averkamp und Co. [APRA98], daß er ein Protokoll spezifiziert, über das zwei CORBA Produkte verschiedener Hersteller kommunizieren können. Das Internet Inter-ORB Protocol benutzt TCP/IP als Transportmedium. Averkamp und Co. zeigen sich verwundert, daß IIOP für eine CORBA-konforme Implementierung nicht vorgeschrieben ist. Dies liegt vermutlich an der Tatsache, daß IIOP erst mit CORBA 2.x eingeführt wurde. Viele ORBs hätten daher zunächst ein proprietäres Protokoll für ihre interne Kommunikation genutzt. Hätten dies jedoch zugunsten einer verbesserten Interoperabiltität zu anderen ORBs aufgegeben.

Zusammenfassend kann aus Anwendersicht folgendes gesagt werden: vom ORB Anbieter wird ihm der ORB Core und der IDL Compiler zur Verfügung gestellt. In der Entwicklungszeit findet aus seiner Sicht folgendes statt: zunächst wird die IDL (die Services und Objekte definiert, die für den Client verfügbar sind) generiert. Der CORBA IDL Compiler generiert nun Client-Stub und Server Skelteton. Anschließend muß im Client und Server Stub noch der Algorithmus umgesetzt werden. Im Gegensatz dazu werden zur Laufzeit werden die Messages zwischen Client und Diensterbringer ausgetauscht.

3 Objekt Adapter

Dieser Abschnitt geht auf die Terminologie ein, welche speziell für das Verständnis des CORBA Objekt Adapters benötigt wird. Außerdem wird auf die Motivation, die hinter dem CORBA Object Adapter steckt eingegangen und auf die Funktionalität, welche von einem CORBA Object Adapter bereitgestellt wird. Außerdem stellt er den Portable Object Adapter (POA) und seinen Vorgänger, den Basic Object Adapter (BOA), vor.

3.1 Terminologie

Definition einiger Bezeichnungen, welche im Zusammenhang mit Objekt Adaptern von besonderer Bedeutung sind [ScVi97]:

CORBA Object: eine“virtuelle”Entität, die von einem ORB lokalisiert werden kann und in der Lage ist Client Anfragen zu erhalten. Ein CORBA Objekt wird mittels seiner Objektreferenz identifiziert, lokalisiert und adressiert.

Diensterbringer: existiert im Zusammenhang mit einem Server und implementiert eine CORBA Methode. In nicht objektorientierten Sprachen wird ein Diensterbringer als eine Zusammenstellung von Funktionen implementiert, welche Daten manipuliert (z.B. eine Instanz eines Structs oder Records), die den Zustand eines CORBA Objektes repräsentieren. In objektorientierten Sprachen sind Diensterbringer Instanzen einer bestimmten Klasse.

Object Id: eine user- oder systemspezifische Kennzeichnung, die dazu dient einem Objekt einen"Namen"zu geben, welcher im Einflußbereich seines Objekt Adapters Gültigkeit hat.

Aktivierung/Deaktivierung: Aktivierung ist der Vorgang, bei dem ein bereits existierendes CORBA Objekt gestartet wird, damit es Anfragen bedienen kann. Da Diensterbringer letzten Endes die Anfragen der Clients bedienen, bedingt Aktivierung, daß das jeweilige CORBA Objekt mit einem passenden Diensterbringer verknüpft wird. Deaktivierung ist konsequenter Weise der Umkehrvorgang zur Aktivierung, die Verknüpfung von CORBA Objekt und Diensterbringer wird hierbei gelöscht, die Existenz des Objektes endet jedoch nicht.

Inkarnation/Etherealisation: Im Zusammenhang mit CORBA Objekten und Diensterbringern bedeutet Inkarnation den Vorgang, bei dem ein Objekt mit einem Diensterbringer verknüpft wird, so daß es Anfragen bedienen kann. Mit anderen Worten: Inkarnation stellt den Vorgang der"Gestaltverleihung"eines virtuellen Objektes dar. Der Umkehrvorgang wird mit Etherealisation bezeichnet.

Active Object Map: eine Tabelle, die von einem Object Adapter gepflegt wird, das seine aktiven CORBA Objekte auf seine Diensterbringer abbildet. Aktive CORBA Objekte werden in dieser Tabelle durch ihre ObjectIds gekennzeichnet.

Die Terminologie, welche hier betrachtet wurde, beschäftigt sich schwerpunktmäßig mit der Lebensdauer von Entitäten in einem CORBA System. Vielleicht ist der einfachste Weg sich den Unterschied zwischen Diensterbringer und CORBA Objects zu merken derjenige, daß ein einzelnes CORBA Object während seiner Lebensdauer durch ein oder mehreren Diensterbringern repräsentiert werden kann. Ebenso kann ein Diensterbringer ein oder mehrere CORBA Objects gleichzeitig repräsentieren. Aktivierung und Deaktivierung beziehen sich ausschließlich auf CORBA Objekte, während sich die Bezeichnungen Inkarnation und Ätherisierung auf Diensterbringer beziehen.

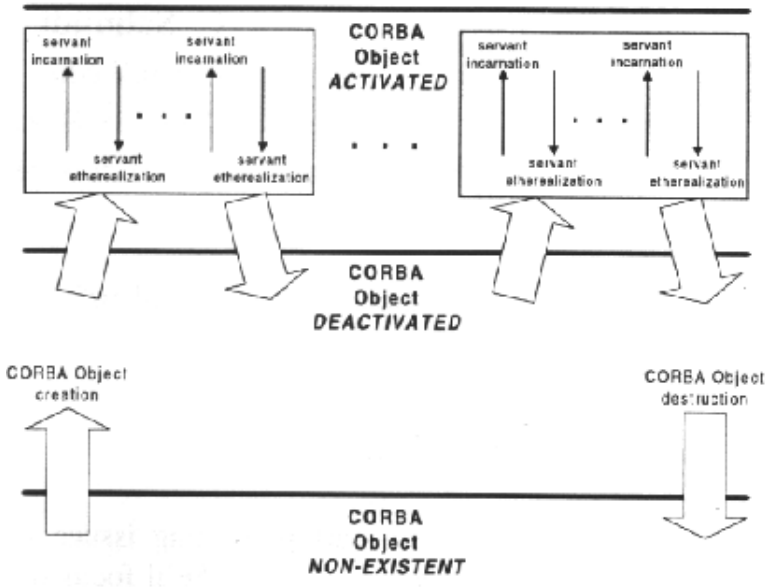


Abbildung 2: Request Lifecycle in the Portable Object Adadpter [ScVi97]

3.2 Motivation

[ScVi97] machen zur Motivation folgende Angaben: Das Referenzmodell zeigt, wie der Object Adapter zwischen ORB Core und statischem bzw. dynamischem Skeleton sitzt. In ihrer Aufgabe als Adapter können unterschiedliche Object Adapter unterschiedliche Arten der Implementierung von Diensterbringern unterstützen. Das Orbix Object Database Adapter Framework erlaubt zum Beispiel, daß Objekte, welche Objektdatenbanken benützen (wie z.B. ODI's ObjectStore), als Diensterbringer für CORBA Objekte verwendet zu werden. Ein anderes Beispiel ist der real-time Object Adapter von TAO.

Es ist sicherlich möglich, eine Architektur für Distributed Object Computing Middleware zu definieren, in der Diensterbringer direkt von einem ORB Core, anstatt von einem Object Adapter registriert werden. Aber eine solche Architektur hätte eine der folgenden Einschränkungen:

Large footprint: Die erste Vorgehensweise würde beinhalten, daß der ORB Core verschiedene Interfaces unterstützen müßte, und daß Dienste unterschiedliche Arten der Implementation von Diensterbringern unterstützen. Dieser Ansatz würde den ORB Core größer, langsamer und komplizierter in Bezug auf Applikationen machen.

Lack of flexibility: Die Alternative wäre, ausschließlich eine Form der Implementierung von Diensterbringern zuzulassen. Zum Beispiel, indem alle Diensterbringer von einer Base Class erben müßten. In diesem Fall wäre jedoch die Nützlichkeit des ORB's, durch Einschränkung seiner Fähigkeit zur Integration (von verschiedenen Programmiersprachen, Arten der Vererbung bzw. Softwaredesigns) beeinträchtigt.

Durch Verwendung von Object Adaptern hat man die Möglichkeit unterschiedlichen Arten der Implementation von Diensterbringern isoliert zu betrachten und zu unterstützen. Hierfür werden dann mehrere Object Adapter verwendet. Das Ergebnis bedeutet einen kleineren und einfacheren ORB Core.

3.3 Funktionalität

Ein CORBA Object Adapter stellt die folgende Funktionalität zur Verfügung [ScVi97]:

Demultiplexing von Anfragen: Object Adapter sind für das Demultiplexing von CORBA Anfragen und Übermittlung an einen geeigneten Diensterbringer zuständig. Wenn der ORB Core eine Anfrage erhält, arbeitet er mit dem Object Adapter mittels eines privaten (d.h. nicht standardisierten) Interfaces zusammen um sicherzustellen, daß eine Anfrage den richtigen Diensterbringer erreicht. Der Object Adapter analysiert die Anfrage, um die Object Id des Diensterbringers zu lokalisieren, welche dazu benützt wird den richtigen Diensterbringer zu lokalisieren und ihm die richtige Anfrage zu übermitteln.

Operationsaufruf: Wenn der Object Adapter den anvisierten Diensterbringer lokalisiert hat, verschickt er die nachgefragte Operation. An diesem Punkt wird das Skeleton benützt um die Parameter in der Anfrage in Argumente umzuwandeln, welche der Diensterbringeroperation übergeben werden.

Aktivierung und Deaktivierung: Object Adapter können CORBA Objekte aktivieren. Im Prozeß können sie auch Diensterbringer inkarnieren um Anfragen für diese Objekte zu bearbeiten. Entsprechend können Object Adapter Objekte deaktivieren und die Etheralisation der zusammenhängenden Diensterbringer übernehmen, wenn diese nicht mehr benötigt werden.

Generierung von Objektreferenzen: Ein Object Adapter ist verantwortlich für die Generierung von Objektreferenzen für die CORBA Objekte, die bei ihm registriert sind. Objektreferenzen identifizieren normalerweise ein CORBA Objekt und beinhalten die Adressinformation, die dazu dient ein Objekt einem verteilten System zu erreichen. Object Adapter arbeiten mit den Kommunikationsmechanismen im ORB Core und dem zugrundeliegenden Betriebssystem zusammen, um sicherzustellen, daß Informationen, welche notwendig sind um ein Objekt zu erreichen in der Objektreferenz enthalten sind. So wird zum Beispiel eine, den Standards des Internet Inter-ORB Protocol (IIOP) entsprechende, Interoperable Object Reference (IOR) die Internet Adresse des Server Hosts enthalten, genauso, wie die Port Number, über den man den Serverprocess erreichen kann.

Object Adapter sind direkt am Versenden von Anfragen an Objektoperationen beteiligt. Deshalb müssen sie sorgfältig entworfen werden, damit sie nicht zu einem Flaschenhals beim versenden von Anfragen werden. Konventionelle ORBs demultiplexen Anfragen der Clients, wie in der folgenden Abbildung dargestellt.

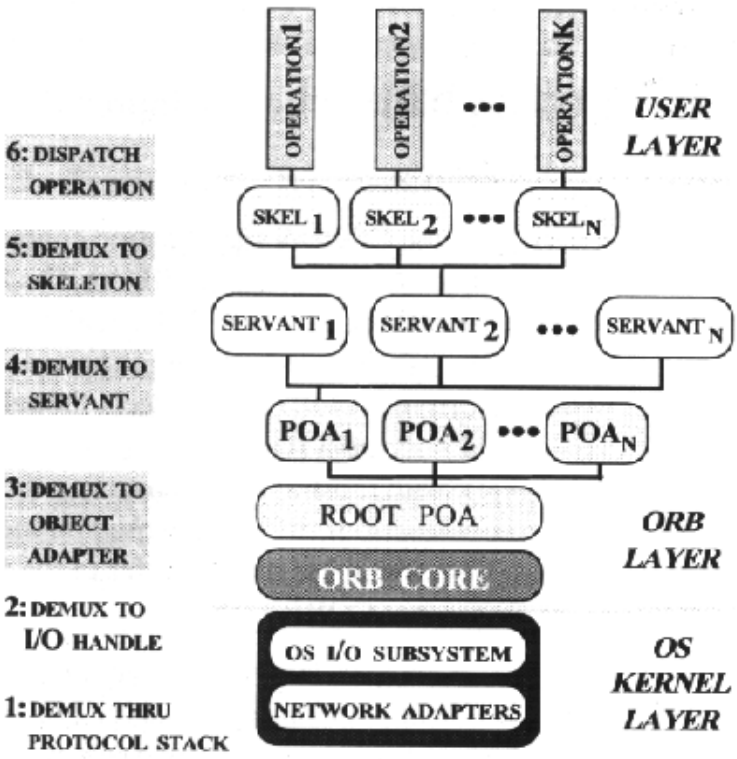


Abbildung 3: Layered CORBA Request Demultiplexing [ScVi97]

Schritte 1 und 2: Der Protocol Stack des Betriebssystems erledigt das mehrfache Demultiplexing der eingehenden Anfragen der Clients, d.h. von der Schnittstellenkarte des Netzwerks, über die Data link (Leitungs-), Network (Vermittlungs-) und Transport Layer (Transportschicht) hinauf zu User/Kernel Boundary und versendet die Daten dann zum ORB Core.

Schritte 3 und 4: Der ORB Core benützt die Adressinformationen im Object Key des Clients um den geeigneten POA und den geeigneten Diensterbringer zu lokalisieren. POAs können hierarchisch organisiert werden. Deshalb können, beim Lokalisieren des POAs welcher den Diensterbringer enthält, mehrere Demultiplex-Schritte durch die Hierarchie notwendig sein.

Schritte 5 und 6: Der POA benützt den Namen der Operation um das geeignete IDL Skeleton zu finden. Dieses wandelt den Anfragepuffer in Operationsparameter um und führt den Aufruf zur Codierung durch, der von den Entwicklern von Diensterbringern bereitgestellt wird.

Demultiplexing von Anfragen von Clients quer durch alle diese Schichten hindurch ist aufwendig, insbesondere wenn eine große Anzahl von Operationen in einem IDL Interface auftauchen und/oder wenn eine große Anzahl von Diensterbringern von einem Object Adapter betreut werden.

3.4 Basic Object Adapter

In der Vergangenheit sind zwei Object Adapter offiziell von der Object Management Group übernommen worden. Die Features und Bewertung des ersten (BOA) soll hier insbesondere dem Verständnis des Designs und der Fähigkeiten des zweiten (POA) dienen.

3.4.1 BOA Features

Die erste Version der CORBA Spezifikation enthielt die Beschreibung des Basic Object Adapters (BOAs). Dieser wird hier nach [ScVi97] beschrieben. Der BOA war dazu gedacht, ein Mehrzweck-Object Adapter zu sein, welcher verschiedene Stile von Diensterbringern unterstützten sollte. Wegen der besagten Flexibilität des BOAs, dachten die original CORBA Architekten, daß es insgesamt nur ein paar Object Adapter (z.B.: einen Bibliotheks Objekt Adapter, einen Datenbank Objekt Adapter, etc.) geben würde. Außerdem dachten sie, daß die meisten Applikationen den BOA verwenden würden.

Der BOA unterstützte folgende vier Modelle für die Aktivierung von Server Processen (nicht CORBA Objekten):

Unshared Server: dies ist ein Server, der nur ein einzelnes CORBA Objekt unterstützt. Ein Unshared Server kann nur Objekte eines Typs aufnehmen. Immer dann, wenn ein neues CORBA Objekt dieses Typs erzeugt wird, startet der ORB automatisch einen neuen Serverprozess um das Objekt aufzunehmen.

Shared Server: dies ist ein Server, der CORBA Objekte häufig verschiedenen Typs unterstützt. Zum Beispiel kann ein solcher Server CORBA Objekte unterstützten, die die Rolle von Factories haben oder von diesen erzeugt wurden. In der Praxis sind die meisten CORBA Server Shared Server.

Persistent Server: dieser Server wird nicht automatisch von einem ORB gestartet. Diese Typen von Servern können zum Beispiel beim Booten des Systems gestartet werden. Die Bezeichnung:“Persistent”ist irreführend, da sie impliziert, daß der Zustand der Diensterbringern auf einem Server automatisch weiterbesteht, unabhängig von Abstürzen und auch dann, wenn das System heruntergefahren ist. Nachdem dies nicht der Fall ist schlagen Schmidt und Vinoski [ScVi97] den Begriff:“manually launched”vor. Desweiteren weisen sie darauf hin, daß der Mechanismus zum starten eines Server Prozesses ein komplett anderer ist, je nach dem ob es sich um einen Shared oder einen Unshared Server handelt. Dies bedeute, daß dieses Aktivierungsmodell sich von den anderen wesentlich unterscheidet.

Server-per-Operation: dieser Typus eines"Servers"stellt nicht nur ein einfacher Prozess dar. Stattdessen, stellt er eine Gruppe von Prozessen dar, von denen jeder einzelne eine gegebene Operation eines bestimmten Typs von CORBA Objekten implementiert. Dieser Typus ist theoretisch für Diensterbringer von Nutzen, welche mit Hilfe von Shell Skripts implementiert wurden. Zum Beispiel mittels verschiedener Shell Skripts welche unterschiedliche Operationen des CORBA Objektes implementieren. Aufgrund der Schwierigkeit diesen Server Stil zu unterstützen und aufgrund seines eingeschränkten Nutzens, unterstützen nur wenige kommerzielle ORBs die Server-per-Operation Variante.

3.4.2 BOA Bewertung

Unglücklicherweise ist die ursprüngliche Spezifikation von CORBA bezüglich des BOA unvollständig. Einige der Schlüsselprobleme [ScVi97]:

- **Kein definierter portabler Weg, um Skeletons mit Diensterbringern zu verknüpfen:**
  Die Spezifikation läßt offen, wie Skeletons aussehen oder wie Diensterbringer mit ihnen verknüpft sind. Eine Folge davon ist, daß die OMG IDL C++ Mapping Specifikation ursprünglich ausschließlich definierte, wie die Bodies von Servantmethoden und die Signaturen der Methoden aussehen sollten. Die Namen der Basisklassen, von denen Diensterbringer erben mußten, wurde nicht spezifiziert. Diese Unterlassung machte das Schreiben von CORBA C++ Code sehr schwierig.

- **Versäumnis, den Vorgang der Registrierung von Diensterbringern zu beschreiben:**
  Implementationen des BOA erlauben Diensterbringer typischerweise implizit (z.B. im Konstruktor des Diensterbringers, wenn dieser inkarniert wird) oder explizit (z.B. durch Aufruf einer Methode des BOA) registriert zu sein. Die original BOA Spezifikation definiert diese APIs jedoch nicht. Deshalb tendiert jeder ORB dazu diese Funktionalität anders zu implementieren.

- **Ignorierung der Problemstellung des Multi-Threadings von Server Prozessen:**
  Multi-Threaded ORBs sind wichtig, da sie den simultanen Ablauf von Aufgaben mit langer Laufzeit ermöglichen, ohne daß dadurch der Ablauf von anderen Aufgaben behindert würde. Nichtsdestotrotz behandelt die CORBA 2.0 BOA Spezifikation dieses Thema nicht und überläßt damit diese Entscheidung den ORB Entwicklern. Ergebnis ist, daß unterschiedliche ORBs Multi-Threading unterschiedlich implementieren.

- **Versäumnis die Funktionen, die nötig sind, damit ein Server auf Anfragen angemessen reagiert, akkurat zu definieren:**
  Der BOA stellt zwei Operationen zur Verfügung, die signalisieren, daß durch ihren Aufruf, der Server anfängt Aufgaben zu bearbeiten: impl_is_ready und obj_is_ready. Für die jeweilige Serveranwendung wird in der BOA Spezifikation das jeweilige Aktivierungsmodell festgelegt. Zum Aktivierungsmodell wird anschließend die passende Methode aufgerufen. Unglücklicherweise ist dieser Teil der Spezifikation sehr unklar, was zur Folge hat, daß unterschiedliche ORB Produkte die zwei Operationen auf sehr unterschiedliche Art verwenden.

[ScVi97] resümieren: Aufgrund der Mängel der BOA Spezifikation hat jeder Hersteller von ORBs die Implementationen auf seine eigene (inkonsistente) Art durchgeführt. Dies hat zum Ergebnis, daß es so gut wie keine Portierbarkeit von Server betreffendem Code zwischen

verschiedenen ORB Produkten gibt. Deshalb hat die OMG einen Request for Papers, zur Verbesserung der Portabilität, in Umlauf gebracht. Hierbei wurde vorgeschlagen, entweder die Probleme des BOAs zu beseitigen oder ihn durch einen neuen Object Adapter zu ersetzen.

3.5 Portable Object Adapter

Im März 1997 wurde der POA bei der OMG eingereicht, um den BOA zu ersetzen. Der POA ermöglicht nun, anders als der BOA, Portablilität für CORBA Serverapplikationen. Außerdem stellt er neue, sehr nützliche Funktionalität zur Verfügung. Selbstverständlich wird es Verkäufern ermöglicht, welche in der Vergangenheit den BOA unterstützt haben, dieses weiterhin zu tun um ihre existierenden Kunden zufriedenstellen zu können [ScVi97]. Ein Versuch den BOA zu korrigieren hätte dazu geführt, daß alle existierenden CORBA-Applikationen, die auf der alten Spezifikation basieren, hätten geändert werden müssen [ScVi98c].

3.5.1 Design Ziele

Designziel (laut [PySc98]) beinhalteten die folgenden Punkte:

Portabilität: der POA erlaubt es Programmierern Diensterbringern zu konstruieren, welche zwischen verschiedenen ORB Implementationen portierbar sind. Folglich kann der Programmierer zwischen verschiedenen ORBs hin und her wechseln, ohne existierenden Servant-Code verändern zu müssen. **Persistente Identitäten (entsprechen globalen Objekten):** der POA unterstützt Objekte, mit persistenten Identitäten. Präziser ausgedrückt, der POA ist designed, Diensterbringer zu unterstützen, welche konstistenten Service für Objekte anbieten, deren Lebensdauer die Lebensdauer von Servern um ein mehrfaches übertreffen.

Automation: Der POA unterstützt transparente Aktivierung von Objekten und implizite Aktivierung von Diensterbringern. Dieser Automatismus erleichtert die Nutzung des POA.

Konservierende Ressourcen: Es gibt eine Vielzahl von Situationen, in denen ein Server CORBA Objekte unterstützen muß. Z.B.: ein Datenbank Server, der jeden Datensatz als ein CORBA Objekt modelliert kann hunderte von Objekten bedienen. Der POA ermöglicht es einem einzelnen Diensterbringer verschieden Object Ids gleichzeitig zu bedienen. Dies ermöglicht es einem Diensterbringer eine Vielzahl von CORBA Objekten gleichzeitig zu bedienen und dabei den Speicherplatz des Servers nur in geringem Umfang zu belasten.

Flexibilität: Der POA ermöglicht es Diensterbringern, vollständige Verantwortung für das Verhalten eines Objektes zu übernehmen. Ein Diensterbringer hat eine Reihe von Möglichkeiten, das Verhalten eines Objektes zu kontrollieren: Er kann sowohl die Identität eines Objektes, als auch die Beziehung zwischen Identität und Zustand eines Objekt festlegen. Außerdem hat der Diensterbringer die Möglichkeit die Speicherung und das Retrieval des Objektzustandes zu steuern sowie Code zur Verfügung zu stellen, welcher als Antwort an Anfragen ausgeführt wird. Schließlich kann er noch jederzeit bestimmen ob ein Objekt existiert oder nicht.

Von Verfahrensweisen/Policies geleitetes Verhalten: Der POA bietet einen erweiterbaren Mechanismus zur Verknüpfung von Verfahrensweisen mit Diensterbringern innerhalb eines POAs. Derzeit unterstützt der POA sieben Verfahrensweisen, u.a. Threading, Retention, Lebensdauer, welche beim Erzeugen des POA ausgewählt werden können. Ein Überblick über diese Verfahrensweisen wird in Kapitel 3.5.5 gegeben.

Nested-POAs: Der POA ermöglicht mehreren unterschiedlichen Instanzen von Nested-POAs in einem Server zu existieren. Jeder POA in dem Server stellt einen Platz für Namen für alle bei ihm registierten Objekte und alle von ihm generierten Child-POAs zur Verfügung. Der POA unterstützt rekursive Löschungen, d.h. das löschen eines POAs löscht alle zugehörigen Child-Prozesse.

SSI und DSI Unterstützung: Der POA erlaubt Programmierern Diensterbringer zu entwerfen, welche zum einen von Static Skeleton Klassen (SSI) erben, welche von OMG IDL Compilern erzeugt wurden, zum anderen von Dynamic Skeleton Interfaces (DSI). Clients benötigen keine Informationen darüber, ob CORBA Objekte von einem DSI oder von einem IDL Diensterbringer unterstützt werden. Bei zwei CORBA Objekte, die das selbe Interface unterstützen, kann der eine von einem DSI und der andere von einem IDL Diensterbringer bedient werden. Des weiteren: ein CORBA Objekt kann zeitweise von einem DSI Diensterbringer bedient werden, auch wenn es ansonsten von einem IDL Diensterbringer bedient wird.

3.5.2 POA Architektur

[PySc98]: Der ORB ist eine Abstraktion, mit der sowohl Client als auch Server direkten Kontakt haben. Im Gegensatz dazu ist der POA eine Komponente des ORB, mit der nur der Server Kontakt hat, d.h. Clients selbst wissen weder, daß der POA existiert noch haben sie Informationen über seine Struktur. Dieses Kapitel beschreibt die Architektur des Modells, welches für das versenden der Anfragen zuständig ist, so wie es vom POA definiert ist und die Interaktionen zwischen seinen Standardkomponenten und dem ORB Core.

Vom Anwender zur Verfügung gestellte Diensterbringer sind beim POA registriert. Clients haben Objektreferenzen mittels derer sie Anfragen stellen, die der POA als Operationen einem Diensterbringer übergibt. ORB, POA und Diensterbringer arbeiten zusammen, um erstens, festzulegen welchem Diensterbringer die Operationen übergeben werden sollten und zweitens, um den Aufruf zu versenden.

Abbildung 4 zeigt die POA Architektur. Wie in der Grafik dargestellt wird vom ORB eine Variante des POA, der sogenannte Root POA erzeugt und gesteuert. Der Root POA steht für Applikationen ständig zur Verfügung. Dies geschieht mit Hilfe des ORB Initialisierungs Interfaces: resolve_initial_references. Der Anwendungsentwickler kann Sevants beim Root POA registrieren, falls die Verfahrensweise des Root POAs, welche in der POA Spezifikation spezifiziert sind für die Anwendung geeignet sind. Für eine Server Anwendung könnte es von Nutzen sein, verschiedene POAs zu erzeugen, um unterschiedliche Arten von CORBA Objekten und/oder unterschiedliche Stile von Diensterbringern zu unterstützen. Z.B. könnte eine Server Applikation zwei POAs haben, von denen jeder unterschiedliche Arten von Objekten unterstützt. Ein Nested POA kann erzeugt werden, indem man die create_POA Factory Operation auf dem Parent POA aufruft. Die Server Anwendung in Abbildung 4 beinhaltet drei andere Nested POAs: A, B und C. POA A und B sind Kinder des Root POA; POA C ist Child von B. Jeder POA hat einen Active Object Table welcher Object Ids auf Diensterbringer abbildet.

Andere Schlüsselkomponenten in einem POA sind:

POA Manager: dieser kapselt die Zustände des Prozesses von ein oder mehreren POAs. Indem Operationen über einen POA Manager aufgerufen werden, können Serverapplikationen Anfragen für verknüpfte POAs warten lassen oder beenden. Außerdem können Applikationen den POA Manager benützen um POAs zu deaktivieren.

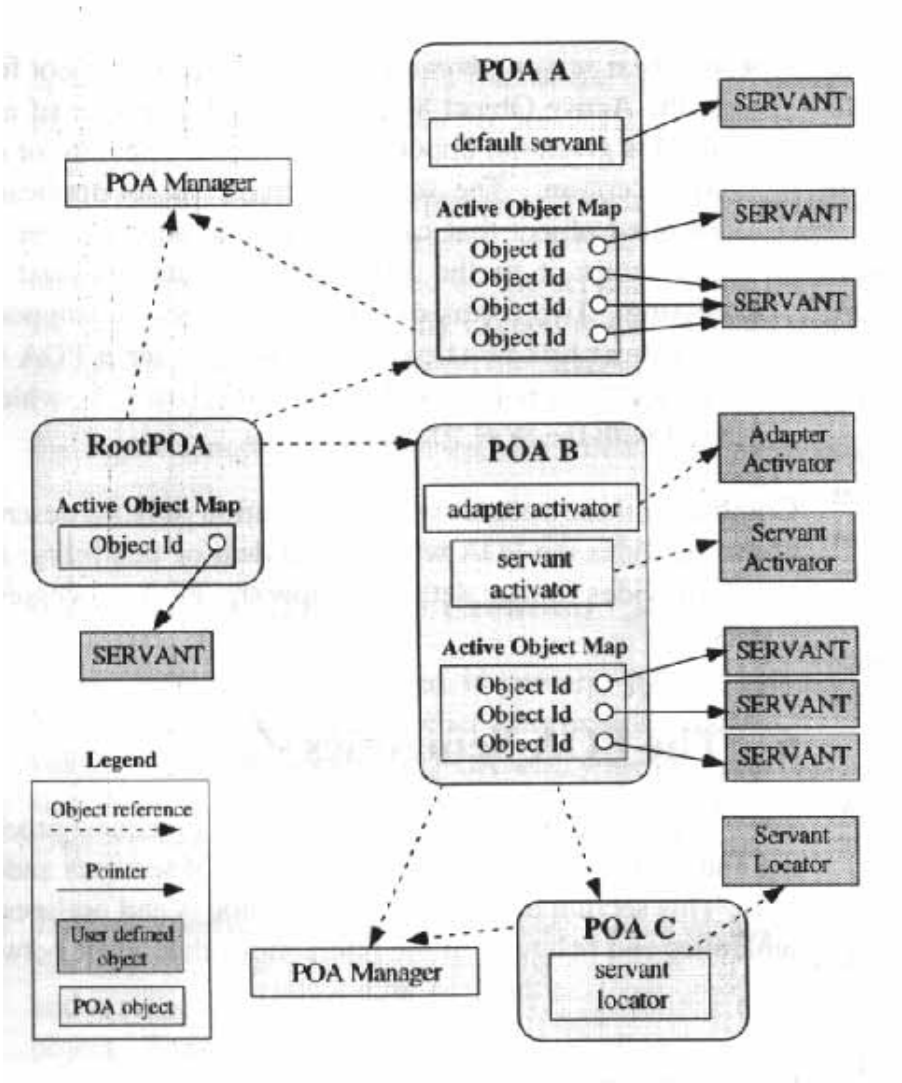


Abbildung 4: POA Architecture [PySc98]

Adapter Activator: ein Adapter Activator kann von einer Anwendung mit einem POA verknüpft werden. Der ORB wird eine Operation auf einem Adapter Activator aufrufen, wenn er eine Anfrage an einen noch nicht existierenden Child POA erhält. Der Adapter Activator kann nun entscheiden, ob er den angefragten POA erzeugt oder nicht. Z.B. wenn die anvisierte Objektreferenz von einem POA erzeugt wurde, dessen voller Name /A/B/C lautet und ausschließlich POA /A und POA /A/B bisher existieren, wird die unknown_adapter operation auf dem Adapter Activator, welcher mit dem POA /A/B verknüpft ist, aufgerufen. In diesem Fall wird POA /A/B als Parent Parameter und C als Name des fehlenden POA an den unknown_adapter übergeben.

Diensterbringermanager: ist ein lokal beschränkter Servant, den Servantapplikationen mit einem POA verknüpfen können. Der ORB nützt Diensterbringermanager um Diensterbringern auf Anfrage zu aktivieren, oder auch zu deaktivieren. Diensterbringermanager sind zum einen dafür verantwortlich, die Verknüpfung eines Objekts mit einem speziellen Diensterbringer zu managen, zum anderen dazu, zu bestimmen, ob ein Objekt existiert oder nicht. Es gibt zwei Typen von Diensterbringermanagern: ServantActivator und ServantLocator. Welcher von beiden in der jeweiligen Situation verwendet wird, hängt von Verfahrensweisen der POAs ab, die im folgenden beschrieben werden. POA Verfahrensweisen/Policies werden in einem eigenen Kapitel (Kapitel 3.5.5) behandelt.

3.5.3 POA Semantik

[PySc98] Der POA hat insbesondere zwei Aufgabent: Die erste beinhaltet die Bearbeitung von Anfragen, die zweite dient der Aktivierung und Deaktivierung von Diensterbringern und Objekten. Dieses Kapitel beschreibt die zwei Methoden sowie Semantik und Art der Zusammenarbeit, welche zwischen den Komponenten der POA Architektur stattfindet.

Request Processing Jede Anfrage eines Clients beinhaltet einen Object Key. Dieser Object Key beinhaltet die Object Id des Target Objects sowie die Identität des POAs der die Target Object Reference erzeugt hat.

Die Bearbeitung einer Anfrage eines Clients läuft in den folgenden Schritten ab:

1. Lokalisierung des Serverprozesses: Wenn ein Client eine Anfrage erstellt, lokalisiert der ORB zunächst einen geigneten Serverprozeß oder verwendet, falls es nötig sein sollte, das Implementation Repository um einen neuen Serverprozeß zu erzeugen. In einem ORB, welcher das IIOP verwendet, identifizieren der Name des Host sowie die Portnummer in der Interoperable Object Referenz (IOR) die Kommunikationsendpunkte des Serverprozesses.

2. Lokalisierung des POA: Sobald der Serverprozess lokalisiert wurde, lokalisiert der ORB den geeigneten POA innerhalb dieses Servers. Wenn der gewählte POA im Serverprozeß nicht existiert hat der Server die Möglichkeit den benötigten POA nochmals zu erzeugen, indem er einen Adapter Activator verwendet. Der Name des gewählten POA ist im IOR spezifiziert, auf einer Art und Weise, über die der Client keine Informationen hat.

3. Lokalisierung von Diensterbringern: Wenn der ORB den geeigneten POA lokalisiert hat, übermittelt er ihm die Anfrage. Der POA findet den geeigneten Servant, indem er seinen Diensterbringer Retention und Request Processing Policies folgt, welche in Kapitel 3.5.5 beschrieben werden.

4. Lokalisierung des Skeletons: Der letzte Schritt, den der POA durchführt, dient der Lokalisierung des IDL Sekeletons, welches die Parameter der Anfrage in Argumente umwandeln wird. Das Skeleton übergibt anschließend diese Argumente als Parameter an die richtige Servantoperation, welche es mittels einer der Operation Demultiplexing Strategies lokalisiert.

5. Handling von Antworten, Ausnahmen (Exceptions) und Location Forwarding: Das Skeleton behandelt Ausnahmen, Rückgabewerte, inout und out Parameter, welche vom Diensterbringer zurückgegeben werden, damit sie an den Client weitergegeben werden können.

Erzeugung von Objektreferenzen Objektreferenzen werden innerhalb eines Servers erzeugt. Objektreferenzen dienen dazu, Object Ids und andere Informationen, welche von einem ORB benötigt werden, einzukapseln. Diese Informationen dienen dem Lokalisieren von Servern und dem POA, mit dem das Objekt verknüpft ist. Z.B. der POA in dessen Bereich die Referenz erzeugt wurde.

Objektreferenzen können von einer Serveranwendung explizit erzeugt werden, wobei hierbei keine Verknüpfung zwischen Objekt und einem aktiven Diensterbringer entsteht. Eine Objektreferenz kann außerdem durch explizite oder implizite Aktivierung von einen Diensterbringer erstellt werden. Wenn eine Referenz von einem Server erzeugt wurde, kann sie auf verschiedene Arten an Clients übergeben werden. Außerdem kann sie als Ergebnis eines Operationsaufrufs zurückgegeben werden. Unabhängig davon, wie eine Objektreferenz erschaffen wurde, kann ein Client, sobald er eine Objektreferenz hat, mit Operationen auf diese Objekt einwirken.

3.5.4 POA Features

[ScVi97] Die POA Spezifikation unterstützt eine große Bandbreite von Funktionen: Anwender- oder systemunterstützte Object Identifiers, persistente und transiente Objekte, explizite und on-demand Aktivierung, verschiedene CORBA Object Mappings für Diensterbringer, vollständige Kontrolle seitens der Anwendung über Verhalten und Existenz der Objekte sowie statische und DSI Diensterbringers. In diesem Kapitel soll nun die Features des POAs noch weiter vertieft werden.

Object Identifiers Eine Object Id kann von der Anwendung oder vom POA zugewiesen werden. Es ist wichtig festzuhalten, daß eine Object Id nicht notwendigerweise zur Identifikation eines Objektes außerhalb des Bereichs seines POAs ausreicht. Das heißt, daß sie nicht global einzigartig sind. CORBA Objekte werden von ihrer Objektreferenz identifiziert, von denen Object Ids nur ein Teil sind. Da Objektreferenzen keine Information für Applikationen darstellen, ist es Clients unmöglich, und gleichzeitig auch unnötig, Object Ids zu analysieren. Clients geben lediglich die Objektreferenzen an, um die Ziele ihrer Anfragen anzugeben. Der ORB erledigt dann den Rest.

Persistente vs. Transiente Objekte Ergänzend zur Lifespan Policy kann festgehalten werden, daß Transiente Objekte in der Regel geringeren Overhead benötigen, da der ORB und der POA keine Information über deren Aktivierung im zeitlichen Verlauf verfolgen müssen. Sie sind geignet für"temporäre"Objekte, wie z.B. Call-Back Objects. Persistente Objekte verändern ihren Zustand in der Regel nicht, im Gegensatz dazu ist der POA weder ein persistentes Objekt, noch stellt er Funktionen zur Verfügung um es Objekten zu ermöglichen ihren Zustand direkt abzuspeichern oder wiederherzustellen.

Aktivierung Serverprozesse standen bei der Aktivierung im Falle des BOA im Zentrum. Im Gegensatz dazu sind die Einrichtungen zur Aktivierung, die der POA bietet, auf die Aktivierung von CORBA Objekten und die Inkarnation von Diensterbringern zugeschnitten:

Explizite Aktivierung: Diensterbringer für CORBA Objekte werden durch direkte Meldung an einen POA registriert. Dies ist für Serverapplikationen sinnvoll, die nur eine begrenzte Anzahl von CORBA Objekten haben.

On-Demand Aktivierung: Registriert einen Diensterbringermanager, den der POA aufruft, wenn er eine Anfrage für ein CORBA Objekt erhält, das noch nicht aktiviert ist. Wenn er einen solchen Aufruf erhält führt der Diensterbringermanager typischerweise eine der folgenden Operationen durch:

1. Falls es notwendig ist führt er die Inkarnation des Diensterbringers durch und registriert ihn beim POA. Dieser sendet anschließend die Anfrage an den Diensterbringer.
2. Er startet eine ForwardRequest Ausnahme mit dem Ziel, die Anfrage an ein anderes Objekt weiterzuleiten.
3. Er startet eine CORBA::OBJECT_NOT_EXIST Ausnahme um anzuzeigen, daß das CORBA Objekt zerstört wurde.

Implizite Aktivierung: Eine Aktion in Bezug auf einen Diensterbringer führt zur Aktivierung, ohne daß der POA dazu explizit aufgefordert würde.

Default Diensterbringer: Die Anwendung registriert einen Default Diensterbringer, der benützt wird, wenn eine Anfrage für ein CORBA Objekt eintrifft, welches noch nicht aktiviert wurde und noch kein Diensterbringermanager registriert wurde. Dieses Feature ist sehr sinnvoll für DSI-basierte Serverapplikationen, da es einem einzelnen DSI Diensterbringer ermöglicht alle CORBA Objekte zu Inkarnieren, ohne daß hierfür ein Diensterbringermanager benötigt würde.

Eindeutigkeit der Object Id Ein POA kann fordern, daß alle CORBA Objekte mit einem eigenen Diensterbringer aktiviert werden. Dies ist eine sehr geradlinige Implementationsvariante, da sie eine eins-zu-eins Beziehung zwischen CORBA Objekten und ihren Diensterbringern beinhaltet. Alternativ kann ein POA zulassen, daß ein Diensterbringerobjekt verschiedene CORBA Objekte inkarniert. Dies ist sehr sinnvoll, um den Bedarf an Serverressourcen zu minimieren.

Verhalten und Existenz von Objekten Applikationen haben vollständige Kontrolle über den Zustand und das Verhalten von Objekten. Da ein POA keinen persistenten Zustand hat, sind Anwendungen verantwortlich für jede benötigte persistenz von Objektzuständen. Anwendungen können feststellen, ob ein Objekt existiert und können eine CORBA::OBJECT_NOT_EXIST Ausnahme erstellen, um anzuzeigen, daß ein Objekt nicht mehr existiert. Desgleichen können sie eine Portable Server::ForwardRequest Ausnahme ausgeben, um dem ORB mitzuteilen, an anderer Stelle nach dem Objekt zu sehen. Die betroffene Client-Anwendung wird bei diesem Forwarding stets mit den für sie relevanten Informationen versehen. POAs unternehmen nichts, um über die persistenten Zustände von CORBA Objekten informiert zu sein, da es eine Vielzahl von Möglichkeiten gibt Persistenz zu implementieren. Falls die POA Spezifikation eine oder mehrere Möglichkeiten um Persistenz zu erreichen vorschreiben würde, bedeutete dies eine übermäßige Limitierung von anderen Klassen der Applikationen für welche eine solchen Vorgehensweise erwiesenermaßen ungeeignet wäre. Überdies wird Objektpersistenz besser von einem higher-level Dienst zur Verfügung gestellt. Der vorhandene Dienst für Objektpersistenz ist zu kompliziert, weshalb die OMG an einem Ersatz für diesen Dienst arbeitet.

Static und DSI Diensterbringer Traditionell basieren die meisten CORBA C++ Serverapplikationen auf der statischen Vorgehensweise. Die Beliebtheit von statischen Skeletons stammt im wesentlichen von ihrer Effizienz, ihrer Bekanntheit unter C++ Programmierern und dem Fehlen des DSI in CORBA bis zur Version 2.0.

Für Serverapplikationen, die die Verwendung von DSI benötigen, wird eine Standard Abstract Base Class: DynamicImplementation (definiert im PortableServer Modul), als Teil der POA Spezifikation zur Verfügung gestellt. Die Verwendung dieser Base Class ähnelt der Verwendung von Skeletons in vererbungsbasierten Diensterbringern, da DSI Diensterbringerklassen von dieser Klasse erben müssen.

3.5.5 POA Policies

[PySc98]: Abgesehen vom Root POA kann der Charakter eines POA, beim Kreieren dem jeweiligen Bedarf angepaßt werden indem zwischen verschiedenen Verhaltensweisen bzw. Policies gewählt wird. Die Verhaltensweisen des Root POAs sind in der POA Spezifikation festgelegt.

Die POA Spezifikation definiert die folgenden Verfahrensweisen:

Threading Policy: mittels dieser Verfahrensweise wird das vom POA verwendete Threading Modell spezifiziert. Ein POA kann entweder Single-Threaded sein oder einen ORB haben, der seine Threads kontrolliert. Single-Threaded bedeutet, daß alle Anfragen sequentiell verarbeitet werden. Falls er Single-Threaded ist, werden alle Anfragen sequentiell abgearbeitet. Bei Multi-Threading werden die Aufrufe des POAs an die Implementation z.B. von Diensterbringer und Diensterbringermanager derart ausgeführt, daß auch durch nicht multithreading-fähigen Code keine Probleme entstehen. Im Gegensatz dazu kann auch eine Threading Policy gewählt werden, bei der der ORB die Threads bestimmt, mittels derer der POA seine Anfragen versendet. Bei Multi-Threading können konkurrierende Anfragen über verschiedene Threads abgearbeitet werden.

Lifespan Policy: Diese Verfahrensweise wird verwendet, um festzulegen, ob ein CORBA Objekt, welches innerhalb eines POA erzeugt wird, persistent oder transient ist. Persistente Objekte können die Prozesse überdauern, in denen sie ursprünglich erzeugt wurden. Im Gegensatz dazu ist dies bei transienten Objekten nicht der Fall. Wenn ein POA deaktiviert wurde, hat jegliche Benutzung einer Objektreferenz, welche für ein transientes Objekt erzeugt wurde, eine CORBA::OBJECT_NOT_EXIST Ausnahme zur Folge. Persistente Objekte sind"normale"CORBA Objekte, die aktiviert werden können, um Anfragen der Clients zu bearbeiten.

Object Id Uniqueness Policy: Diese Verfahrensweise dient dazu, festzulegen, ob die im POA aktiven Diensterbringer, einzigartige Object Ids haben müssen. Die Politik erlaubt es der Anwendung zu kontrollieren, ob ein Diensterbringer mit nur einem, oder mit verschiedenen CORBA Objekten verknüpft werden kann, um für diese Anfragen zu bedienen.

Object Id Alignment Policy: wird benützt um festzulegen, ob Objekt IDs im POA von der Anwendung oder vom ORB generiert wurden. Falls der POA auch die Persistent Lifespan Policy hat, müssen mit dem ORB verknüpfte Object IDs fü alle Instanzen des gleichen POA eindeutig sein.

Implicit Activation Policy: Diese Verfahrensweise wird verwendet um festzulegen, ob implizite Aktivierung von Diensterbringern im POA unterstützt wird. Ein C++ Server kann einen Diensterbringer erzeugen. Anschließend kann er, durch festlegen seines POAs und unter Verwendung seiner _this Methode, den Diensterbringer registrieren und in einem einzigen Arbeitsschritt eine Objektreferenz erstellen.

Diensterbringer Retention Policy: Hier wird festgelegt, ob der POA Aktive Diensterbringer mittels einer Active Object Map festhält. Entweder hält ein POA eine feste Verknüpfung zwischen einem Diensterbringer und CORBA Objekten oder er stellt für jede eingehende Anfrage eine neue CORBA Objekt/Diensterbringer Verknüpfung her.

Request Processing Policy: Diese Verfahrensweise dient dazu, zu spezifizieren, wie Anfragen im POA durchgeführt werden sollen. Wenn eine Anfrage für ein gegebenes CORBA Objekt eingeht, kann der POA: 1. Seine Active Object Map zu Rate ziehen (Wenn die Object Id in der Active Object Map nicht gefunden wird schickt der POA eine CORBA::OBJECT_NOT_EXIST Meldung zurück zum Client). 2. Einen Default Diensterbringer verwenden (falls die Object Id nicht in der Active Object Map gefunden wurde und ein Default Diensterbringer zur Verfügung steht). 3. Einen Diensterbringermanager einschalten. (Falls die Object Id in der Active Object Map nicht gefunden wurde, hat der Diensterbringermanager, falls vorhanden, die Möglichkeit einen Diensterbringer zu lokalisieren oder eine eine Ausnahme zu vermelden. Der Diensterbringermanager ist ein von der Anwendung bereitgestelltes Objekt, welches einen Diensterbringer inkarnieren oder aktivieren kann. Außerdem hat er die Möglichkeit einen Diensterbringer dem POA

zurückzuschicken um die kontinuierliche Anfragenbearbeitung sicherzustellen. Zwei Formen von Diensterbringermanagern werden unterstützt: Diensterbringeraktivierer, für Server mit Retain Policy und DiensterbringerLocator, für Non-Retain Policy.

Die Kombination dieser Verfahrensweisen mit den Retention Policies, welche oben beschrieben werden unterstützen nachhaltig die Flexibilität des POA. Das nächste Kapitel gibt weitere Details darüber, wie der POA Anfragen bearbeitet.

3.5.6 Verwendung des POAs zur Entwicklung eines einfachen Servers

[ScVi97]: Die Flexibilität, welche durch die POA Policies zur Verfügung gestellt werden, mögen im ersten Moment komplex oder gar überwältigend erscheinen. Der POA kann jedoch auch auf unkomplizierte Art für einfache Applikationen genützt werden. Hier ist z.B. ein einfaches und portierbares Server Programm.

```
Int main (int argc, char *argv[])
{
using namespace CORBA;
using namespace PortableServer;

// Initialize the ORB.
ORB_var orb = ORB_init (argc, argv);

// Obtain an object reference for the Root POA.
Object_var obj = orb->resolve_initial_references (‘‘RootPOA’’);
POA_var poa = POA::_narrow(obj);

//Incarnate a servant.
My_Servant_Impl servant;

// Implicitly register the servant with the RootPOA.
obj = servant._this ();

// Start the POA listening for requests.
poa->the_POAManager ()->activate ();

// Run the ORB’s event loop.
Orb->run ();

// ...
}
```

Dieses Beispiel initialisiert zunächst den ORB, dann erhält es vom ORB eine Objektreferenz des Root POAs. Anschließend erzeugt es die Instanz einer Diensterbringerklasse, My_Servant_Impl und ruft seine _this Methode auf. Dies registriert implizit den Diensterbringer mit dem Root POA und generiert eine Objektreferenz für das neu erzeugte CORBA Objekt. Wegen der Default Policies, welche für den Root POA von der Portability Enhancement Specifikation vorgeschrieben werden, ist dieses neue CORBA Objekt ein transientes Objekt. Der POA Manager ist nun aktiviert um Anfragen bedienen zu können. Zum Schluß wird die ORB::run Operation aufgerufen, um den Main Event Loop, des zu der Server Anwendung gehörenden ORBs, zu starten. Dieser Main Event Loop bedient alle Anfragen und leitet die Aufrufe an My_Servant_Impl weiter.

3.5.7 Programmierung eines persistenten CORBA Objektes mit dem POA

[ScVi98c] Wenn der Serverprozeß aus dem vorhergehenden Kapitel beendet ist, ist das CORBA Objekt, welches von dem Programm erzeugt wurde nicht mehr vorhanden. Dies liegt daran, daß es als ein transientes Objekt erzeugt wurde. Transiente Objekte haben eine Lebensdauer, die an die Lebensdauer des Prozesses, in dem sie erzeugt wurden gebunden sind. Schlüssel zum Verständnis von transienten Objekten ist die Tatsache, daß, falls unser ursprüngliches Serverprogramm nochmals gestartet würde, würde es ein neues transientes Objekt, mit einer anderen Objektreferenz erzeugen. Deshalb würden Clients, welche noch Referenzen des vorhergehenden CORBA Objektes verwenden eine CORBA::OBJECT_NOT_EXIST Ausnahme erhalten, falls sie versuchen sollten auf dieses Objekt über die alte Referenz zuzugreifen. Nichtsdestotrotz gibt es CORBA-Applikationen, die man beenden und wieder starten können muß. Diese benötigen persistente Objekte. Ein Beispiel hierfür ist das Root NamingContext Objekt.

Damit unser Server ein persistentes Objekt statt eines transienten Objektes enthält sind zwei Änderungen notwendig:

1. Die Objektreferenz darf nicht mittels des Root POAs erzeugt werden: die Lifespan Policy des Root POAs ist transient, nicht persistent. Deshalb muß ein anderer POA, mit einer persistenten Lifespan Policy erzeugt und verwendet werden.

2. Das CORBA Objekt darf nicht implizit aktiviert werden: Unser Aufruf der _this Methode auf unserem Diensterbringer bewirkt, daß der POA die ObjectID unseres Objektes für uns generiert. Wir müssen die Object Id explizit kontrollieren, um sicher zu gehen, daß bei jeder Aktivierung des Serverprozesses die gleiche Id verwendet wird.

Unser Beispiel muß nun wie folgt aussehen:

```
Int main (int argc, char **argv)
{
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

// Obtain an object reference for the Root POA.
CORBA::Object_var obj = orb->resolve_initial_references (‘‘RootPOA’’);
PortableServer::POA_var poa = POA::_narrow(obj);

// (1) Create the desired POA Policies.
CORBA::PolicyList policies;
policies.length (2);
policies[0] = poa->create_lifespan_policy (PortableServer::PERSISTENT);
policies[1] = poa->create_id_assignment_policy (PortableServer::USER_ID);

// (2) Create a POA Manager for persistent objects.
PortableServer::POAManager_var poa_mgr = poa->the_POAManager();
poa = poa->create_POA (‘‘persistent’’, poa_mgr, policies);

// (3) Create an ObjectId.
PortableServer::ObjectId_var oid = string_to_ObjectId (‘‘my_object’’);

// Create a servant to service client requests.
```

```
Null_Servant_Impl servant;

// (4) Register the servant with the POA explicitly.
poa->activate_object_with_id (oid, &servant);

// Allow the POA to listen for requests.
poa_mgr->activate ();

// Run the ORB's event loop.
orb->run ();

// ...
}
```

Die vier wichtigsten Modifikationen:

1. Kreierung der gewünschten POA Policies: Persistente Lebensdauer (durch Aufruf der POA Policy Factory Operation create_lifespan_policy mit dem Argument: PERSISTENT) bzw. User-supplied ObjectId (anstatt dem POA zu erlauben, einen Object Identifier zur Verfügung zu stellen, um das Objekt innerhalb dieses POAs zu identifizieren, wurde hier die Möglichkeit gewählt unsere eigene ObjectId zur Verfügung zu stellen. Auch diese Verfahrensweise wird mittels der POA Policy Factory Operation gewählt, nämlich indem die create_id_assignment_policy mit dem USER_ID Enumeral aufgerufen wird.)

2. Kreierung eines POAs für persistente Objekte: Alle POAs mit Ausnahme des Root POAs werden als Kinder von anderen POAs erzeugt und sind, von der Logik her, unterhalb ihrer Parent POAs angesiedelt. Unser POA wird durch Aufruf der create_POA Methode über den Root POA erzeugt. Dieser Methode wird ein Name für den neuen POA mitgegeben ("persistent"), ein POAManager (z.B. der des Root POA) und eine Liste mit Policies.

3. Kreierung einer ObjectId: Kreierung einer PortableServer::ObjectId für das persistente CORBA Objekt. Die ObjectId identifiziert das Objekt eindeutig innerhalb des Bereiches des POAs, bei dem es registriert ist. Sie ist nicht global eindeutig.

4. Explizite Registrierung des Diensterbringers mit dem POA: Die letzte Veränderung beinhaltet die explizite Aktivierung des CORBA Objektes. Dies geschieht durch Aufruf der activate_object_with_id Methode auf dem POA und Übergabe der ObjectId und eines Pointers auf den Diensterbringer. Intern speichert der POA die ObjectId und den Diensterbringer Pointer in seiner Active Object Map, welche für das anschließende Demultiplexing und Versenden von Anfragen verwendet wird.

3.5.8 Kreierung einer persistenten Objektreferenz

[ScVi98c]: Wichtig ist festzuhalten, daß die activate_object_with_id Operation keine Objektreferenz erzeugt. Sie registriert nur den Diensterbringer beim POA und reaktiviert das CORBA Objekt. Somit stellt unser Programm bisher nur einen Platz zur Verfügung, indem das bereits existierende CORBA Objekt reaktiviert werden kann.

Punkt (3) muß wie folgt verändert werden:

```
PortableServer::ObjectId_var oid = string_to_ObjectId (‘‘my_object’’);

// Check command-line arguments to see if we’re creating the object or
// reactivating it.
If (argc == 2 && strcmp (argv[1],‘‘create’’) == 0)
{
// Create an object reference.
CORBA::Object_var obj = poa->create_reference_with_id
(oid,‘‘IDL:Null:1.0)’’;

// Stringify it and write it to stdout.
CORBA::String_var s = orb->object_to_string (obj);
cout << s.in () << endl;
}
else
{
// Create a servant to service client requests.
Null_Servant_Impl servant;

// Same reactivation code as before.
poa->activate_object_with_id (oid, &servant);

// Allow the POA to listen for requests.
poa_mgr->activate ();

// Run the ORB’s event loop.
orb->run ();
}
```

Diese Veränderung ermöglicht es dem Serverprogramm, abhängig davon, wie es aufgerufen wird, die Rolle eines Factory Programms oder eines Servers zu übernehmen. Wenn wir dieses Programm mit der Kommandozeile“create”aufrufen, erzeugt es unser CORBA Objekt unter Verwendung der POA::create_reference_with_id Operation.

Mit dieser Vorgehensweise wird kein separates Administrative Factory Programm, zum erzeugen der Objekte, benötigt. Sie reduziert außerdem den, für Wartungsaufgaben benötigten Overhead. Wichtiger noch ist, daß ein einzelnes kombiniertes Factory/Server Programm das Problem eliminiert, verfolgen zu müssen, welches Factory Programm bei welchem Server registriert werden sollte. Weitere Details und C++ Code zu Diensterbringerklassen sowie Diensterbringermanagern sind in [ScVi98b] bzw. [ScVi98a] enthalten.

3.5.9 POA Bewertung

OMG’s Portable Object Adapter Spezifikation definiert eine große Bandbreite von Standard Policies, welche es Entwicklern ermöglichen ein, für verschiedenste Anwendungsbeispiele, maßgeschneidertes Verhalten eines ORBs zu erreichen. Auch wenn der POA eine sehr junge Ergänzung des OMG CORBA Modells darstellt, so kann er doch auf die Erfahrung der Anwender und Designer des Basic Object Adapters (BOA) und anderer Object Adapter aufbauen. Grundsätzlich ist der POA mächtiger und portierbarer als der BOA [PySc98].

Ein mögliches Problem mit den Policies des POAs ist die Tatsache, daß kommerzielle ORB Anbieter wahrscheinlich die zur Verfügung stehenden Policies durch ihre eigenen Policies ergänzen werden und somit die Portierbarkeit gefährden könnten. Um dies zu verhindern wäre es

wichtig, daß die Anbieter von ORBs auch weiterhin ihre Ideen bei der OMG einreichen, damit daraus weitere nützliche POA Policies entstehen können [ScVi97]. In ihrem Artikel [ScVi98b] welcher im Juni 1998, ein Jahr nach der Spezifikation des POAs und vier Monate später als die CORBA Spezifikation 2.2, erschienen ist geben die Autoren die folgende Prognose ab: Auch wenn die Vorgehensweise beim POA sich von den meisten derzeitigen Implementationen unterscheidet, bietet er bessere Konsistenz, Sicherheit und Skalierbarkeit. Wir erwarten, daß die meisten kommerziellen CORBA Anbieter ihre ORBs in Kürze soweit bringen werden, daß sie die POA Spezifikation erfüllen. Averkamp und Co. [APRA98] geben in ihrem Artikel an, daß bisher leider nur wenige Hersteller den POA anbieten, einige jedoch bereits implementiert haben, obwohl ihr aktuelle ORB-Version 'nur' CORBA 2.0 oder 2.1 unterstützt. Außerdem sind einige Anbieter dabei, den POA zu implementieren. Wenn man Entwicklungszeiten in Betracht zieht könnte man vermuten hier läge eine reine Verzögerung bei der Umsetzung vor. Der POA mag aber auch für manche Anbieter unattraktiv sein, da er, laut Averkamp und Co. [APRA98], nur eingeschränkt für Echtzeitanwendungen geeignet ist.

Echtzeitfähigkeit beschreibe jedoch nur eine von vielen Dienstgüte- oder "Quality of Service"-Anforderungen (QoS), die in einem verteilten System von Bedeutung wären. Eine wichtige QoS-Anforderung sei z.B. die Ausfallsicherheit. Aktuelle Forschung beschäftige sich daher mit der Integration allgemeiner QoS-Mechanismen in CORBA. Man spricht in diesem Zusammenhang auch von "QoS-enabled CORBA".

Auf der anderen Seite verbessert die Verwendung des POAs die Portabilität von CORBA-Applikationen und damit die Unabhängigkeit von einem bestimmten Hersteller. Gerade diese Unabhängigkeit von einem speziellen einen Hersteller kam bei der Entwicklung des CORBA-Standards laut Averkamp und Co. [APRA98] durch die OMG besondere Bedeutung zu. Als Vorraussetzung hierfür gilt der spezifizierte Begriff der "CORBA-Konformität". Um außerdem mögliche Abhängigkeiten, die durch proprietäre Erweiterungen seitens des jeweiligen Herstellers entstehen können, zu reduzieren, ist die Vergabe eines, an strenge Maßstäbe gebundenen Zertifikats für konforme ORBs durch die OMG geplant.

Literatur

[APRA98] P. Averkamp, A. Puder, K. Römer und K. Auel. Urbi et ORBi - von Big Blue bis GPL: Object Request Broker. *iX Magazin für Professionelle Informationstechnik*, Oktober 1998.

[PySc98] I. Pyarali und D. C. Schmidt. An Overview of the CORBA Portable Object Adapter. *Special issue of the ACM StandardView Magazines on CORBA*, Dezember 1998.

[SaCu98] W. Sadiq und F. Cummins. *Developing Business Systems with CORBA.* Cambridge University Press, Cambridge. 1998.

[ScVi97] D. C. Schmidt und S. Vinoski. Object Adapters: Concepts and Terminology. *SIGS C++ Report* Band 11, November 1997.

[ScVi98a] D. C. Schmidt und S. Vinoski. C++ Servant Managers for the Portable Object Adapter. *SIGS C++ Report* Band 14, September 1998.

[ScVi98b] D. C. Schmidt und S. Vinoski. Developing C++ Diensterbringer Classes Using the Portable Object Adapter. *SIGS C++ Report* Band 13, Juni 1998.

[ScVi98c] D. C. Schmidt und S. Vinoski. Using the Portable Object Adapters for Transient and Persistent CORBA Objects. *SIGS C++ Report* Band 12, April 1998.

Echtzeit CORBA

Alexander Schmid

Kurzfassung

CORBA ist ein wichtiger Standard für Middlewarearchitekturen, der die Entwicklung von verteilten Anwendungen und Diensten vereinfacht. Um echtzeitfähige Applikationen entwickeln zu können, die strenge Dienstgüteanforderungen stellen, benötigt man eine CORBA-Middleware der nächsten Generation. Ein zentraler Bestandteil von CORBA ist der ORB. In diesem Seminarbericht werden zuerst die Schwächen der bisherigen ORB-Implementierungen aufgezeigt. Daraufhin soll am Beispiel von TAO, einem echtzeitfähigen ORB, gezeigt werden, wie man diese Schwächen beseitigt. Es werden Optimierungen des ORB-Kernels, des I/O-Subsystems, der Demultiplexstrategien und der Dienstgütespezifikation aufgezeigt. Abschliessend werden Messergebnisse von statischen und dynamischen echtzeitfähigen CORBA-Systemen diskutiert.

1 Einleitung

Verteilte Anwendungen gewinnen, aufgrund ihrer grossen Leistungsfähigkeit, immer mehr an Bedeutung. CORBA (Common Object Request Broker Architecture) ist ein wichtiger Standard für das objektorientierte verteilte Programmieren. Er ermöglicht die einfache Interaktion zwischen verteilten Client- und Serverobjekten. Ziel von CORBA ist es, eine Integrationsplattform zur Realisierung verteilter Objekte zu schaffen, die unabhängig ist von Rechnerarchitekur, Programmiersprache und Betriebsystem.

Im Zentrum von CORBA, wie auch bei anderen Middlewaresystemen, steht der Object Request Broker (ORB). Der ORB automatisiert die wichtigsten Aufgaben, wie die Objektlokalisierung, die Objektaktivierung und die Fehlerbehebung (s. Bild 1). Im ORB befinden sich die Stubs und Skeletons, diese Komponenten sind für die Datenrepräsentation (OSI-Schicht 6) in der CORBA-Architektur verantwortlich. Sie transformieren typisierte Operationsparameter von einer anwendungsabhängigen Repräsentationsebene in eine plattformunabhängige Repräsentationsebene. Dies wird marshaling (verpacken) genannt und umgekehrt demarshaling (entpacken) genannt. Zum Beispiel kann mittels marshaling eine 16-bit Integerzahl durch eine allgemeine Integerzahl dargestellt werden. Im ORB findet auch das Demultiplexen der Clientanforderungen für den Server statt. Hierbei werden einzelne Informationen, wie z.B. die Clientadresse, aus der Clientanforderung ausgelesen.

Ein immer grösser werdender Teil an verteilten Applikationen benötigt Dienstgütegarantien. Dazu gehören Simulationen, Telekommunikations-, Multimedia- und Echtzeitsysteme. Bei den Echtzeitsystemen kommt es im wesentlichen darauf an, eine Anforderung in einer vorgegebenen Zeit garantiert bearbeiten zu können. Es geht also primär um die sichere Diensterbringung in einer garantierten Zeit und sekundär um die Geschwindigkeit.

Die bisherigen CORBA-ORBs können die geforderten Dienstgütegarantien, wie Ende-zu-Ende Prioritätserhaltung, strenge obere Schranken für Latenzzeit- und Jittergarantien bzw. strenge untere Schranken für Bandbreitegarantien, aus Gründen die in Kapitel 2 erläutert werden,

nicht geben. Deshalb müssen für Echtzeitanwendungen eine Reihe von Optimierungen an den ORBs durchgeführt werden, wie sie im Kapitel 3 u. 4 erläutert werden. Unter anderem muss der Objekt Adapter, der ORB-Kern und das I/O-Subsystem optimiert werden, um die wichtigsten Eigenschaften von Echtzeitanwendungen, wie die Vorhersagbarkeit und Skalierbarkeit, zu unterstützen.

Im wesentlichen werden die Optimierungen an statischen Anwendungen gezeigt, bei denen die Dienstgüteanforderungen vor Beginn der Ausführung feststehen. Bei diesen Anwendungen gibt es gute Fortschritte und Erfahrungen. Ein Beispiel für eine statische Anwendung, ist das Rate Monotonic Scheduling, bei dem die Anforderungen in einem festen Takt ausgeführt werden.

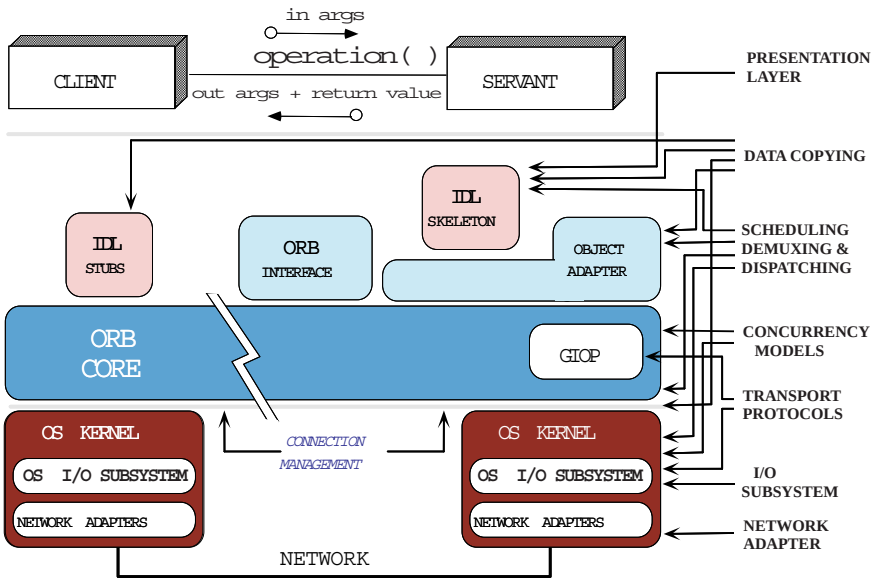


Abbildung 1: Eigenschaften und Optimierungen von Echtzeit-ORB-Endsystemen [Doug98]

2 Begrenzte Einsatzfähigkeit von CORBA für Echtzeitanwendungen

Die bisherigen Erfahrungen mit CORBA in der Telekommunikation und in der medizinischen Bildverarbeitung zeigen, dass es gut für Anwendungen geeignet ist, die nur "bestmögliche" Dienstgüteanforderungen stellen. Jedoch reichen die konventionellen CORBA-Implementierungen für harte Echtzeitanforderungen aus folgenden Gründen nicht aus.

2.1 Fehlende Spezifikation der QoS-Schnittstellen

Der CORBA 2.X Standard unterstützt keine Schnittstellen, um QoS Anforderungen für Ende-zu-Ende Verbindungen zu beschreiben. Zum Beispiel gibt es keinen Standardweg für Clients um festzustellen, welche relative Priorität ihre Anforderungen an den ORB haben. Die Clients können dem ORB auch nicht mitteilen in welchen Zeitabständen dieser Operationen, die z.B. eine periodische Prozess-Deadline besitzen, ausführen soll.

Auch sieht der CORBA Standard keine Möglichkeit vor, in den Anwendungen eine Zugangskontrolle zu implementieren. Zum Beispiel würde man für einen Videoserver mit einer bestimmten Bandbreite die Anzahl der Clients beschränken und weitere Clients ablehnen, um jedem einzelnen Client beste Videoqualität garantieren zu können. Andererseits würde man

für einen Aktienkursservice eine sehr grosse Anzahl Clients zulassen und die ganze Bandbreite auf alle gleichmässig aufteilen.

2.2 Fehlende Durchsetzung der Dienstgüte

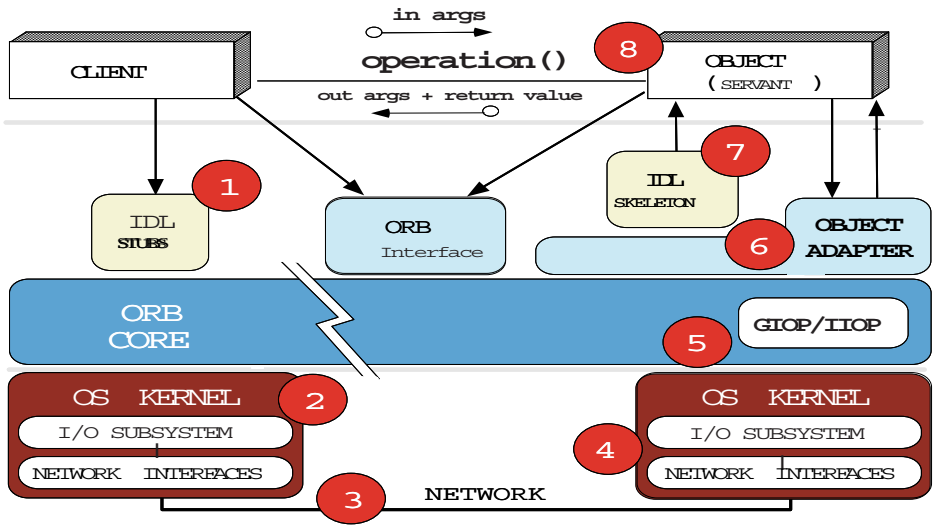
Die herkömmlichen ORBs unterstützen keine Ende-zu-Ende Dienstgüteerzwingung, z.B. von einer Applikation zur anderen über ein Netzwerk. Die meisten ORBs übertragen, planen und senden ihre Clientanforderungen nach dem FIFO-Prinzip. Doch FIFO-Strategien können Priority Inversions hervorrufen, die auftreten wenn eine Anforderung mit niederer Priorität, die Ausführung einer Anforderung mit höherer Priorität blockiert. Desweiteren können die Applikationen bei Standard ORBs die Prioritäten der Threads, welche die Anforderungen bearbeiten, nicht festgelegt werden. Auch gibt es keine Betriebsmittelkontrolle. So ist es möglich, dass ein einzelner Client die gesamte Netzbandbreite und ein sich falsch verhaltender Server die gesamte CPU für sich in Anspruch nehmen kann.

2.3 Fehlende Eigenschaften für die Echtzeitprogrammierung

In der CORBA Spezifikation braucht man keinen ORB, um Clients zu benachrichtigen, wenn auf der Transportschicht Flusskontrolle durchgeführt wird. Auch werden keine zeitgetakteten Operationen unterstützt. Deshalb ist es schwierig, portable und effiziente Echtzeitanwendungen zu entwickeln, die sich auch dann noch deterministisch verhalten, wenn das ORB-Endsystem oder Netzwerkressourcen zeitweise unereichbar sind.

2.4 Fehlende Leistungsoptimierung

Konventionelle ORB-Endsysteme verursachen überflüssige Latenzzeit. Desweiteren erzeugen sie viele Priority Inversions und sind Quellen von Indeterminismus wie Abbildung 2 zeigt.



1) CLIENT MARSHALING
2) CLIENT PROTOCOL QUEUEING
3) NETWORK DELAY
4) SERVER PROTOCOL QUEUEING
5) THREAD DISPATCHING
6) REQUEST DISPATCHING
7) SERVER DEMARSHALING
8) METHOD EXECUTION

Abbildung 2: Quellen für Latenzzeit und Priority Inversions in konventionellen ORBs [Doug98]

Die Overheads stammen von nicht optimierten Datenrepräsentationsschichten, die Daten unnötig kopieren und verschieben und somit den Prozessor Cache überlasten. Desweiteren gibt es interne Pufferstrategien, die kein einheitliches Verhalten mehr bei verschiedenen Nachrichtengrössen aufweisen und ineffiziente Demultiplexalgorithmen.

3 Verbesserungen

Am Beispiel von TAO, dem Echtzeit-ORB der Universität von Washington, sollen einige Verbesserungen mit Hinblick auf die Echtzeitfähigkeit von CORBA untersucht werden.

3.1 Hochleistungs- und Echtzeit-I/O-Subsysteme

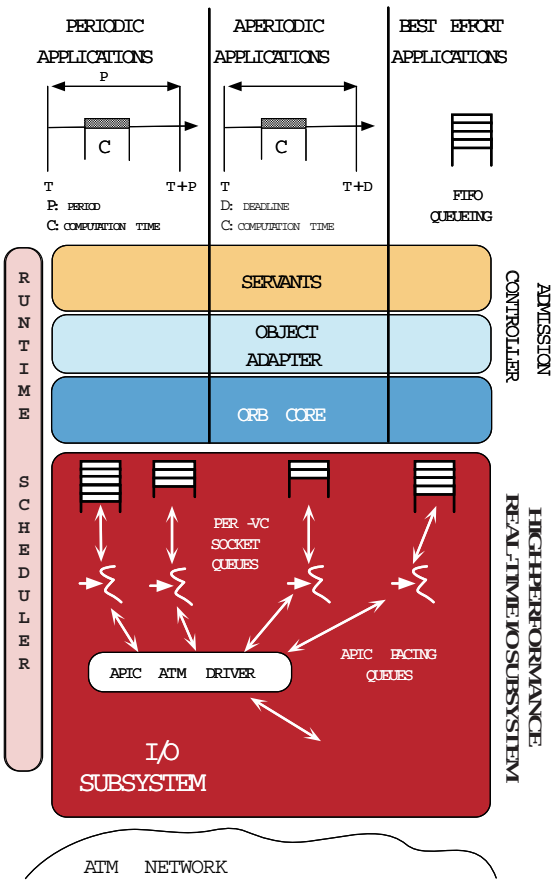


Abbildung 3: Komponenten des TAOs Echtzeit-I/O-Subsystems [Doug98]

Das I/O-Subsystem ist dafür verantwortlich, dem ORB und den Anwendungen den Zugriff auf low-level Netzwerk- und Betriebsystemressourcen zu ermöglichen, wie z.B. Gerätetreiber, Protokollstacks und Prozessoren. Die Herausforderungen bei der Entwicklung eines I/O-Subsystems sind, es den Anwendungen zu ermöglichen ihre Dienstgüteanforderungen spezifizieren zu können, die Einhaltung der Dienstgütespezifikationen zu unterstützen, d.h. Priority Inversions und Indeterminismus zu minimieren. Desweiteren soll der ORB-Middleware erlaubt werden, die Dienstgüteeigenschaften der darunterliegenden Netzwerkressourcen und Betriebsystemressourcen in verschiedene Dienstgüteebenen einzuteilen. Die Komponenten des TAO Echtzeit-Netzwerk-I/O-Subsystem werden in Abbildung 3 gezeigt. Sie beinhalten (1) eine Hochgeschwindigkeits-ATM-Netzwerkschnittstelle (2) ein Hochleistungs-Echtzeit I/O-

Subsystem (3) einen Echzeit Scheduling Service und einen Run-Time-Scheduler und (4) eine Zugangskontrolle. Auf einzelne Komponenten wird im folgenden eingegangen.

3.1.1 Echtzeit I/O Subsystem

TAO ist eine Erweiterung des Streammodells, das von Echtzeitbetriebsystemen wie Solaris, VxWorks und Lynx OS unterstützt wird. Um Priority Inversions zu vermeiden, wird bei TAO ein Pool aus Kernel-Threads im voraus alokiert, die für die Protokollverarbeitung bestimmt sind. Diese Kernelthreads laufen unter der gleichen Priorität wie die Anwendungthreads. Um eine vorhersagbare Leistung zu gewährleisten, gehören diese Kernelthreads zu einer Echzeit-I/O-Schedulingklasse. Diese Schedulingklasse benützt Rate-Monotonic-Scheduling, für die Unterstützung von periodischen Echtzeitprozessen.

Wenn ein Echtzeit-I/O-Thread durch den Betriebssystemkernel aufgenommen wurde, dann ist TAOs RIO-Subsystem dafür verantwortlich, dass erstens die Priorität dieses Threads relativ zu den anderen Threads dieser Klasse berechnet wird und zweitens der Thread periodisch verarbeitet wird, damit er seine Deadlines einhalten kann.

3.1.2 Echtzeit-Scheduling-Service und Run-Time-Scheduler

TAOs Realtime Scheduling Service ist ein objektorientierter Ansatz, für die Spezifikation der Anforderungen der Prozesse, wie z.B. die Verarbeitungszeit C, Periode P und Deadline D. Wenn alle Operationen in den Zeitplan passen, wird vom Scheduling Service jeder Anforderung eine Priorität zugewiesen. Zur Laufzeit werden diese Prioritätszuweisungen von TAOs Run-Time-Scheduler gebraucht. Der Run-Time-Scheduler bildet die einzelnen Clientanforderungen auf Prioritäten ab, die vom Threadverarbeiter des lokalen Endsystems verstanden werden können. Dieser Threadverarbeiter garantiert die Einhaltung der Prioritäten der Echtzeit-I/O-Threads. Ausserdem kann mit seiner Hilfe vorherbestimmt werden, ob der Zeitplan zur Laufzeit auf dem entsprechenden Endsystem eingehalten werden kann.

3.1.3 Zugangskontrolle

Damit die Dienstgüteanforderungen der Anwendungen sicher erfüllt werden, unterstützt TAO eine Zugangskontrolle, für seine Echtzeit-I/O-Schedulingklasse. Diese Zugangskontrolle erlaubt es dem Betriebsystem, entweder die spezifische Verarbeitungszeit eines Threads zu garantieren oder diesen abzulehnen. Die Zugangskontrolle, die von grosser Bedeutung ist, ist für Echtzeitsysteme mit deterministischen und/oder statistischen Dienstgüteanforderungen geeignet.

3.2 Effiziente und vorhersagbare Object Adapter

Der Object Adapter ist in der CORBA-Architektur dafür verantwortlich, den Server mit einem ORB zu verbinden, ankommende Clientanforderungen für den Server zu demultiplexen und die entsprechende Operation dieses Servers aufzurufen. Die eigentliche Herausforderung bei der Entwicklung eines Object Adapters für Echtzeit-ORBs ist das effiziente, vorhersagbare und skalierbare demultiplexen von Clientanforderungen.

Man kann den Object Adapter von TAO mittels perfect hashing oder aktivem demultiplexen (vgl. Kap.3.2.2) so konfigurieren, dass Clientanforderungen direkt in O(1) auf Serveroperationen (oder auch Operationstupel) abgebildet werden können.

3.2.1 Konventionelle ORB-Demultiplex-Strategien

Eine Standard GIOP-Client-Anforderung enthält die Identität des Remote-Objekts und die Art der Remote-Operation. Ein Remote-Objekt wird durch einen sogenannten "Object Key", einer Oktetreferenz, repräsentiert. Eine Remote-Operation wird durch einen String repräsentiert. Konventionelle ORBs demultiplexen die Clientanforderungen zu den entsprechenden Serveroperationen mittels einer geschichteten Demultiplexarchitektur, bei der die Clientanforderungen mehrmals einen Demultiplexer durchlaufen.

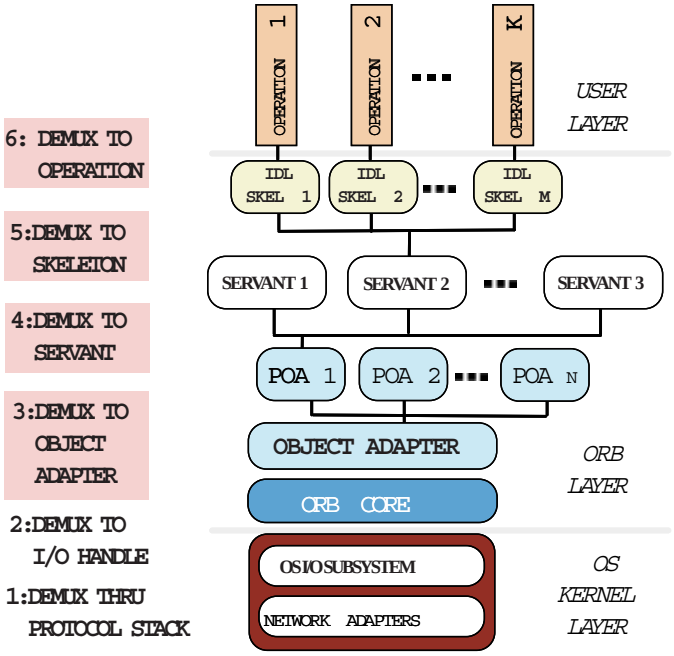


Abbildung 4: Geschichtetes Demultiplexen der CORBA-Anforderungen[Doug98]

Zum Beispiel bewerkstelligt der Protokollstack des Betriebsystems, das mehrmalige Demultiplexen der hereinkommenden Clientanforderungen, z.B durch die Data-Link-Schicht, das Netzwerk und die Transportschichten hoch zum User/Kernel und dem ORB-Kernel.

Jedoch ist ein geschichtetes Demultiplexen generell ungeeignet für Hochleistungs- und Echtzeitanwendungen und zwar aus folgenden Gründen.

- Schlechte Effizienz
  Das Demultiplexen der Klientanforderungen durch die vielen verschiedenen Schichten kostet viel Zeit, speziell wenn eine grosse Anzahl an Operationen in einem IDL-Interface auftritt und/oder viele Server von einem Objekt Adapter bedient werden.

- Zunehmende Priority Inversions und Indeterminismus
  Geschichtetes Demultiplexen kann Priority Inversions verursachen, denn auf Dienstgüteinformationen der Serverschicht können Gerätetreiber der unteren Schicht und der Protokollstack im I/O-Subsystem eines ORB-Endsystem, nicht zugreifen. Deshalb demultiplext der Object Adapter nach der FIFO-Strategie, also nach der Ankunftsreihenfolge. Bei der FIFO-Strategie kann es passieren, dass Pakete mit höherer Priorität für eine unbestimmte Zeit warten müssen, während niederpriore Pakete demultiplext und verarbeitet werden. Herkömmliche CORBA-Implementierungen verursachen einen erheblichen Demultiplexoverhead. Es gibt Untersuchungen [Anir99a] herkömmlicher ORBs, die etwa 17% der Aufrufdauer im Server mit der Verarbeitung von Demultiplexanforderungen verbringen.

3.2.2 TAO's optimierte Demultiplexstrategien

Um die Beschränkungen konventioneller ORBs zu beseitigen, verfügt TAO über Demultiplexstrategien, die in Abbildung 5 verdeutlicht werden.

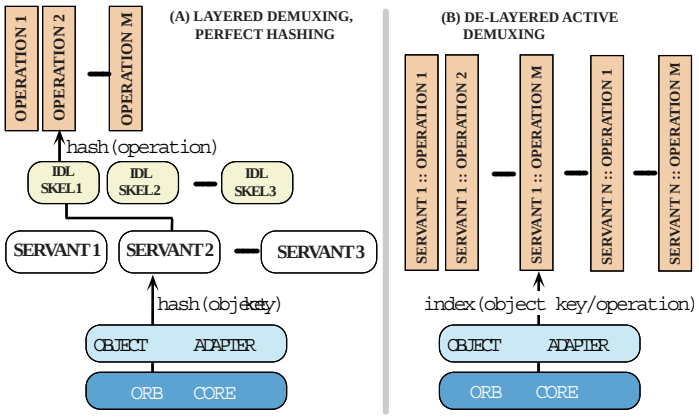


Abbildung 5: Optimiertes Demultiplexen der CORBA-Anforderungen [Doug98]

- Perfektes Hashen
  Die Strategie des perfekten Hashens, die in Abbildung 5(A) gezeigt wird, ist eine zweistufige, geschichtete Demultiplexstrategie. Die Implementierung benützt eine automatisch generierte perfekte Hashfunktion, um den Server zu lokalisieren. Eine zweite perfekte Hashfunktion wird benutzt, um die Operation zu lokalisieren. Der wesentliche Vorteil dieser Strategie besteht darin, dass Server- und Operationssuche einen Zeitaufwand im schlechtesten Fall von O(1) besitzen.

  TAO benützt GNU's gperf-Tool [Referenz seite 13] um perfekte Hashfunktionen für Objekt-Keys und Operationsnamen zu generieren, dazu müssen aber die Keys a priori bekannt sein. In vielen deterministischen Echtzeitsystemen können die Server und die Operationen statisch konfiguriert werden. Für diese Art von Applikationen ist es möglich, mittels perfektem Hashen die Server und Operationen zu lokalisieren.

- Aktives Demultiplexen
  TAO unterstützt auch eine eher dynamische Demultiplexstrategie, die aktives Demultiplexen genannt wird und in Abbildung 3(B) zu sehen ist. Bei dieser Strategie durchläuft der Client einen Objekt-Key, der direkt den Server und die Operation im schlechtesten Fall in O(1) identifiziert. Der Client erhält diesen Objekt-Key, wenn er eine Referenz auf ein Serverobjekt erhält, z.B. durch einen Nameservice oder einen Tradingservice. Wenn eine Anforderung beim Server-ORB ankommt, dann benützt der Objekt Adapter den Objekt-Key und den CORBA-Paketkopf, um den Server und seine assoziierten Operationen in einem einzigen Schritt zu identifizieren.

Im Gegensatz zum perfekten Hashen müssen bei der aktiven Demultiplexstrategie nicht alle Objekt-Ids a priori bekannt sein. Dies macht es geeigneter für Anwendungen, die CORBA-Objekte dynamisch verwalten. Sowohl perfektes Hashen als auch aktives Demultiplexen können Clientanforderungen effizient und vorhersagbar demultiplexen und dies unabhängig von der Anzahl der aktiven Verbindungen und der Anzahl der Operationen, die in der IDL-Schnittstelle definiert sind.

3.3 Effiziente und vorhersagbare ORB-Kernel

Der ORB-Kernel ist die Komponente in der CORBA-Architektur, die die Transportverbindungen kontrolliert, Clientanforderungen an den Objektadapter liefert und eventuelle Antworten an die Clients zurückliefert. Der ORB-Kernel beinhaltet typischerweise das ORB-Demultiplex-Modell und das Parallelitätsmodell.

Die wichtigsten Herausforderungen bei der Entwicklung eines ORB-Kernel sind erstens, die Implementierung einer effizienten Protokollverarbeitung für CORBA Inter-ORB-Protokolle wie GIOP und IIOP, zweitens die Bestimmung eines passenden Verbindungs- und Parallelitätsmodells, das die gesamte Arbeitskapazität der ORB-Endsystemkomponenten vorhersagbar aufteilen kann, über Operationen eines oder mehrerer kontrollierter Threads und drittens sollte der ORB-Kernel leicht adaptierbar sein für neue Endsystem- oder Netzwerkumgebungen, sowie für neue Dienstgüteanforderungen der Anwendungen. Die nächsten Abschnitte beschreiben, wie TAOs ORB-Kernel entworfen wurde, um diesen Ansprüchen gerecht zu werden.

3.3.1 TAOs Inter-ORB-Protokoll Engine

TAOs Protokoll Engine ist eine hochoptimierte Echtzeitversion von SunSofts IIOP Referenzimplementierung, die in ein Hochleistungs-I/O-Subsystem implementiert ist. Dadurch kann TAOs ORB-Kernel mit dem Client, Server und jedem dazwischenliegenden Knoten zusammenarbeiten, um die Anforderungen gemäss den Dienstgüteanforderungen zu verarbeiten. Das Design erlaubt es Clients die relative Priorität ihrer Anforderungen zu bestimmen und erlaubt TAO die Durchsetzung der Ende-zu-Ende-Dienstgüte-Anforderungen.

Im folgenden gehen wir auf die bestehenden CORBA Interoperability-Protokolle ein und beschreiben wie TAO die Protokolle effizient und vorhersagbar implementiert.

3.3.2 Überblick über GIOP und IIOP

CORBA wurde so entwickelt, dass es über mehreren Transportprotokolle laufen kann. Das Standard-ORB-Interoperability-Protokoll ist unter dem Namen General Inter-ORB Protokoll (GIOP) bekannt. GIOP unterstützt ein standardmässiges Ende-zu-Ende Interoperability-Protokoll zwischen potentiell heterogenen ORBs. GIOP spezifiziert ein abstraktes Interface, das auf Transportprotokolle abgebildet werden kann, die bestimmte Anforderungen erfüllen. Zum Beispiel müssen die verwendeten Protokolle verbindungsorientiert sein, eine zuverlässige Nachrichtenübermittlung und einen nicht typisierter Bytestrom bereitstellen. Bei einem ORB, der GIOP unterstützt kann jede Applikation den ORB zum Senden oder Empfangen von Standard-GIOP-Nachrichten benützen. Die GIOP-Spezifikation besteht aus den folgenden Elementen:

- Common Data Representation(CDR) Definition

  Die GIOP Spezifikation definiert eine allgemeine Datenrepräsentation (CDR). CDR ist eine Transfersyntax, die die OMG-IDL-Typen der Endsystemherstellerformate auf ein Format abbildet, welches sowohl little-endian als auch big-endian als binäre Datenformate unterstützt. Die Daten werden CDR-kodiert über das Netzwerk transferiert.

- GIOP Nachrichtenformate

  Die GIOP Spezifikation definiert Nachrichten zum Senden von Anforderungen, Empfangen von Antworten, Lokalisieren von Objekten und das Verwalten von Kommunikationskanälen.

- GIOP Transportmöglichkeiten

  Die GIOP Spezifikation beschreibt auch die Arten der Transportprotokolle, welche GIOP Nachrichten übermitteln können. Zusätzlich wird in der GIOP Spezifikation beschrieben, wie Verbindungen verwaltet werden und es werden einzuhaltende Bedingungen für die Nachrichtenreihenfolge definiert.

Das CORBA Inter-ORB-Protokoll (IIOP) ist eine Abbildung von GIOP auf die TCP/IP Protokolle. ORBs die das IIOP verwenden sind dazu in der Lage mit anderen ORBs zu Kommunizieren. Denn durch IIOP wird das interoperable objekt referenz (IOR) Format definiert, in dem die Ortsangabe der ORBs enthalten ist.

3.3.3 Effiziente und vorhersagbare Implementierung von GIOP/IIOP

Im CORBA 2.x Standard unterstützen weder GIOP noch IIOP die Spezifizierung oder Durchsetzung der Ende-zu-Ende Dienstleistungsgüteanforderungen für Anwendungen. Dies macht GIOP/IIOP ungeeignet für Echtzeitanwendungen, welche die Latenzzeit und den Jitter von TCP/IP Protokollen nicht tolerieren können. Desweiteren bereiten die Routingprotokolle, wie IPv4 mit fehlenden Funktionen wie z.B. Paketzugangskontrolle oder Ratenkontrolle Schwierigkeiten, was zu erheblicher Überlastung und zu nichteingehaltenen Deadlines in Netzwerken und Endsystemen führt.

Um diese Mängel auszugleichen, bietet TAOs ORB-Kernel eine prioritätsbasierte Nebenläufigkeitsarchitektur, eine prioritätsbasierte Verbindungsarchitektur und ein Echtzeit Inter-ORB Protokoll (RIOP) an, auf die im weiteren Verlauf eingegangen wird.

- TAOs prioritätsbasierte Nebenläufigkeitsarchitektur

  TAOs ORB-Kernel kann so konfiguriert werden, dass für jede Anwendungsprioritätsebene ein Echtzeitthread zur Verfügung gestellt wird. Jeder Thread in TAOs ORB-Kernel kann mit einem Reaktor verbunden werden, der gewisse Reaktoreingenschaften beinhaltet, um das Demultiplexen und Event-handler-Ausführungen flexibel und effizient zu gestalten.

  Wenn TAOs Reaktoren beim Server eingesetzt werden, dann demultiplexen sie die ankommenden Clientanforderungen für die Verbindungshandler, die die GIOP Verarbeitung verrichten. Diese Handler Arbeiten mit TAOs Objekt Adapter zusammen, um die Anforderungen für die Anwendungsebene der Serveroperationen vorzuverabeiten. Die Operationen können sowohl in der Priorität des Clients, der die Operation aufgerufen hat, als auch in der Priorität des Echtzeit ORB-Kernel-Threads, der die Operation empfangen hat, ausgeführt werden. Die letztere Möglichkeit ist gut geignet für deterministische Echtzeitapplikationen, da sie Priority Inversions und Indeterminismus in TAOs-ORB-Kernel minimieren. Desweiteren wird das Kontextwechseln und der Synchronisationsoverhead reduziert, weil der Serverstatus nur dann gelockt werden muss, wenn ein Server zwischen den verschiedenen Threadprioritäten interagiert. TAOs prioritätsbasierte Nebenläufigkeitsarchitektur ist optimiert für statisch konfigurierte Anwendungen, die feste Prioritäten besitzen. TAO ist gut geeignet für Zeitplanungs- und Analysetechniken, welche die Priorität mit gewissen Raten verbinden, wie zum Beispiel rate monotonic scheduling (RMS) oder rate monotonic analysis(RMA). Manche Anwendungen führen ihre Operationen in Ratengruppen aus, wobei unter einer Rate der Kehrwert der Frequenz verstanden wird. In einer Ratengruppe werden alle periodischen Prozessoperationen zusammengefasst, die in bestimmten Raten vorfallen (z.B. 20Hz,10Hz,5Hz und 1Hz) und einem Pool von Threads zugewiesen, der eine Zeitplanung mit festen Prioritäten nutzt.

- TAOs prioritätsbasierte Verbindungsarchitektur
  Bild4 zeigt wie TAO mit einer prioritätsbasierten Verbindungsarchitektur konfiguriert werden kann. In diesem Modell erhält jeder Clientthread einen Connector in einem threadspeziefischen Speicher aufrecht. Jeder Connector verwaltet einige vorher eingerichtete Verbindungen zu den Servern. Für jede Threadpriorität wird eine separate Verbindung im Server-ORB aufrechterhalten. Diese Anordnung erlaubt es den Clients ihre Ende-zu-Ende-Prioritäten zu erhalten, auch wenn die Anforderungen ORB-Endsysteme und Kommunikationsverbindungen durchqueren.

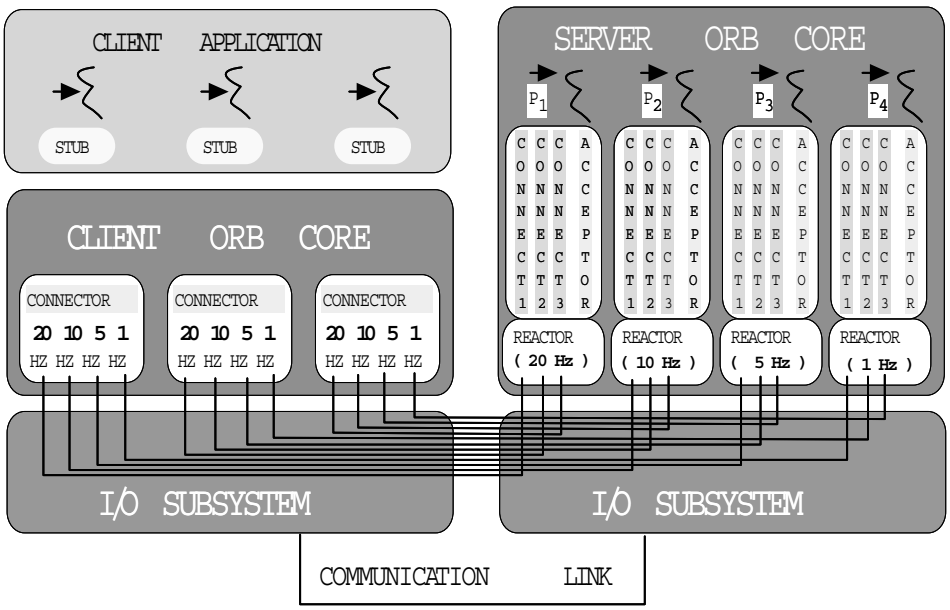


Abbildung 6: TAOs prioritätsbasierte Verbindungs- und Nebenläufigkeitsarchitektur[Doug98]

Bild6 zeigt auch wie der Reaktor jeder einzelnen Threadpriorität in einem Server-ORB, so konfiguriert werden kann, dass er einen Akzeptor nutzen kann. Ein Akzeptor ist ein Socketendpunkt, der an einer bestimmten Portnummer wartet, ob ein Client eine Verbindung zu einer ORB-Instanz aufbauen will. Diese ORB-Instanz läuft in einer bestimmten Threadpriorität. Bei TAO kann jede Prioritätsebene ihren eigenen Akzeptorport besitzen, z.B. bei statisch geplanten Anwendungen können die Anforderungen den 20Hz, 10Hz, 5Hz und 1Hz Ratengruppen zugeordnet werden. Anforderungen, die an diesen Socketports ankommen, können dann von den entsprechenden Echtzeitthreads, die feste Prioritäten besitzen, verarbeitet werden. Wenn ein Client eine Verbindung aufgebaut hat, dann erstellt der Akzeptor eine neue Socketwarteschlange im Server-ORB und einen GIOP-Verbindungsverwalter der diese Wartschlange bedient. TAOs I/O-Subsystem benützt die Portnummer, welche mit den ankommenden Anforderungen mitgeliefert wird, als ein Demultiplexschlüssel zur Verbindung der Anforderungen mit der entsprechenden Socketwarteschlange. Diese Anordnung verhindert Priority Inversions durch das ORB-Endsystem, das die an den Netzwerkschnittstellen ankommenden Anforderungen mit den entsprechenden Echtzeitthreads des Zielservers assoziiert. Somit wird mit dem frühen Demultiplexen in TAO (vgl. Kapitel3.2.2), die Dienstgüte des ORB-Endsystems von der Netzwerkschnittstelle zu den Anwendungsdienstleistern, vertikal integriert.

3.3.4 TAOs Echtzeit Inter-ORB-Protokoll (RIOP)

TAOs Strategie, für jede Verbindung eine Priorität bereitzustellen, wie sie oben erläutert wurde, ist für Anwendungen mit festen Prioritäten optimiert, die ihre Anforderungen in ihren

jeweiligen Zeitabständen über feste statische Verbindungen übertragen und mit der Priorität des Echtzeitthreads des Servers verarbeitet werden. Anwendungen die eine dynamische Dienstgüte benötigen oder die Priorität des Clients auf den Server übertragen möchten, benötigen ein flexibleres Protokoll.

Für diese Fälle bietet TAO das Echtzeit Inter-ORB-Protokoll(RIOP) an. RIOP ist eine Implementierung des GIOP, die es ORB-Endsystemen erlaubt, ihre Dienstgüteattribute von Client zu Server Ende-zu-Ende zu übertragen. Als wichtiges Beispiel kann TAOs RIOP Abbildungsmechanismus, die Priorität jeder Operation Ende-zu-Ende mit einer GIOP-Nachricht transferieren. Das empfangende ORB-Endsystem benutzt dieses Dienstgüteattribut, um die Priorität des Threads zu setzen, der die Operationen auf Serverseite ausführt.

Um die Kompatibilität mit existierenden IIOP-basierenden ORBs zu gewährleisten, wird bei RIOP die Dienstgüteinformation im "service-context" übertragen. Der "service-context" ist eine Datenstruktur im GIOP-Paket, die transparent übertragen wird und nicht zwingend ausgewertet werden muss.

Um es nocheinmal zu verdeutlichen, RIOPs service-context, der mit jedem Clientaufruf mitgeliefert wird, beinhaltet Attribute, welche die Dienstgüteparamter der Operation beschreiben. Diese Attribute beinhalten Priorität, Ausführungsperiode und die Kommunikationsklasse, wie z.B. Isochron für kontinuierliche Medien, Burst für grosse Datenmengen, Message für kleine Nachrichten mit geringen Verzögerungszeitanforderungen und Message-Stream für Nachrichtensequenzen, die in einem gewissen Takt bearbeitet werden müssen.

Zusätzlich zum Transport der Dienstgüteattribute der Clients, ist TAOs RIOP in der Lage, CORBA GIOP auf eine Reihe von Netzwerken abzubilden, unter anderem Hochgeschwindigkeitsnetzwerke wie ATM LANs und ATM/IP WANs. Für Anwendungen, die nicht die ganze Zuverlässigkeit von RIOP benötigen, kann RIOP einzelne Transportschichtfunktionen bereitstellen und direkt auf virtuellen Verbindungen im ATM laufen. Dies ist zum Beispiel für Telefonkonferenzen oder bei bestimmten Bildübertragungen , die keinen bestätigten Dienst oder eine Fehlererkennung auf Bitebene benötigen, sinnvoll.

3.4 Effiziente und vorhersagbare Stubs und Skeletons

Marshaling und Demarshaling stellen grosse Flaschenhälse in Hochleistungskommunikationssubsystemen dar, denn sie beanspruchen während sie auf Daten zugreifen oder diese kopieren, einen grossen Teil der CPU-Leistung, des Speichers und der I/O-Bus-Ressourcen. Deshalb ist es äusserst wichtig, dass man für Echtzeit-ORBs eine effiziente (Daten-) Repräsentantionsschicht entwickelt, die es erlaubt verpacken und entpacken vorhersagbar durchzuführen und gleichzeitig kostspielige Operationen wie dynamische Speicherverteilung und das Datenkopieren vermeidet.

Bei allen CORBA-ORBs wird die (Daten-)Repräsentationsschichtverarbeitung auf der Clientseite durch Stubs und auf Severseite durch Skeletons durchgeführt, welche automatisch von einem hochoptimerten IDL-Compiler erstellt werden. TAOs IDL-Compiler unterstützt verschiedene Strategien, um die IDL-Datentypen zu verpacken bzw. entpacken. Zum Beispiel kann TAOs IDL-Compiler kompilierte und/oder interpretierte IDL-Stubs und IDL-Skeletons generieren. Diese Eigenschaft erlaubt es nun den Applikationen zu wählen, zwischen intepretierten Stubs/Skeletons, die etwas langsamer sind aber dafür eine kompaktere Grösse haben und den kompilierten Stubs/Skeletons, die schneller sind aber mehr Speicherplatz benötigen.

Desweiteren kann TAO auch vorab verpackte Anwendungsdateneinheiten (ADUs, applikation data units), die immer wieder benötigt werden, zwischenspeichern. Dieses Zwischenspeichern verbessert die Leistung, wenn die ADUs sequentiell transferiert werden, zum Beispiel in einer

"Anforderungsschleife", bei der sich zusätzlich die ADUs von Übertragung zu Übertragung nur wenig unterscheiden. In einem solchen Fall ist es nicht notwendig jedesmal die gesamte Anforderung zu verpacken. Diese Optimierung benötigt einen Echtzeit-ORB der eine Flussanalyse des Applikationscodes vornehmen kann, um herausfinden zu können welche Anforderungsfelder zwischengespeichert werden können. Eine solche Flussanalyse kann nur unter gewissen Umständen gemacht werden, z.B. bei Anwendungen die mit einem statistischen Echtzeitservice auskommen, oder wenn die Szenerien des schlechtesten Falles genügend Zeit lassen, um die Deadlines trotzdem einzuhalten.

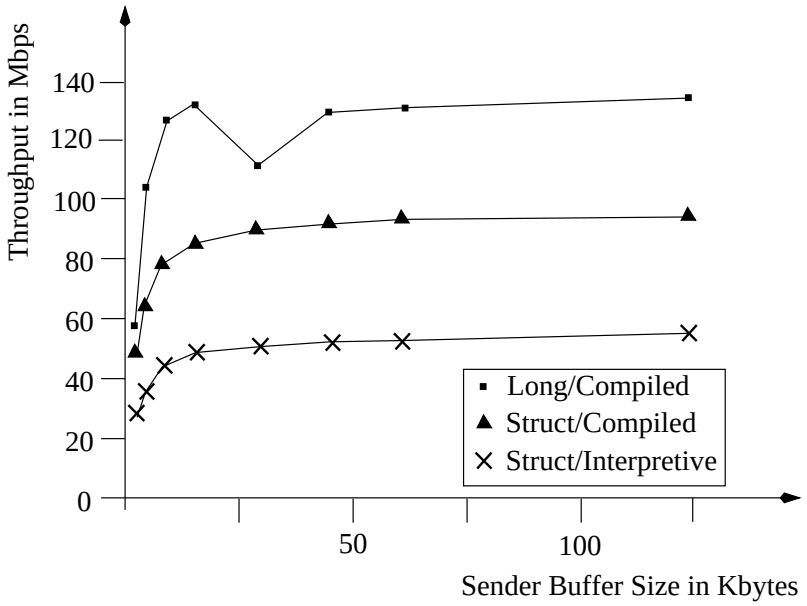


Abbildung 7: Durchsatzvergleich: TAOs interpretiertes vs. kompiliertes Verpacken (Entpacken)[Anir99b]

4 Spezifikation der Dienstgüteanforderungen in TAO

Dienstgüteverbesserungen auf der Netzwerk- und Betriebssystemebene erfüllen nicht die Eigenschaften, wie sie die übliche objektorientierte Middleware fordert. Die wesentliche Arbeit beim Hinzufügen von Dienstgüteunterstützung an ORB-Middleware besteht darin, den Mechanismus der Dienstgüteverarbeitung auf der Netzwerk- und Betriebssystemebene und deren inneren Zusammenhänge auf das objektorientierte Programmiermodell abzubilden. Dieser Abschnitt beschreibt das von TAO benützte objektorientierte Echzeitprogrammiermodell. Bei TAO kann die Dienstgüte für jede Operation spezifiziert werden, hierfür wird das Echtzeit-IDL-Schema von TAO benützt.

4.1 Überblick über die Dienstgütespezifikation bei TAO

In nichtverteilten, deterministischen Echtzeitsystemen ist die CPU-Kapazität die knappste Ressource. Deshalb muss die benötigte CPU-Bearbeitungszeit für die Abarbeitung der Clientanforderung a priori bestimmt werden, damit die CPU-Kapazität entsprechend reserviert werden kann. Bei TAO müssen deshalb CPU-Kapazitätsanforderungen bei TAOs Offline Scheduling Service angemeldet werden. TAOs Dienstgüteanforderungen können auch auf weitere gemeinsam genutzte Ressourcen ausgedehnt werden, wie z.B. Anforderungen an das Netzwerk und die Busbandbreite, die durch Anwendung von Analyseverfahren und durch Messungen entstehen.

Im folgenden soll gezeigt werden, wie TAOs Diestgütespezifikationen beim CPU-Scheduling für IDL-Operationen aussehen, die Echtzeitoperationen beinhalten. Hierfür verfügt TAO über die beiden Echtzeit-IDL (RIDL) Schemen: Die "RT-Operation" Schnittstelle und die "RT-Info" Struktur. Diese beiden Schemen übertragen Dienstgüteinformationen an den ORB für jede einzelne Operation. Dieser Weg der Dienstgütespezifikation soll dem Anwendungsprogrammierer entgegenkommen, da sie direkt auf das objektorientierte Programmiermodell aufsetzt.

4.1.1 Die RT-Operation Schnittstelle

Die RT-Operation Schnittstelle ist ein Übertragungsmechanismus für die CPU-Anforderungen , der zu den Anwendungsoperationen gehörenden laufenden Tasks, an TAOs Scheduling Service, wie sie im folgenden CORBA IDL Interface gezeigt wird. Wie in Abbildung 8 gezeigt wird enthält die RT-Operation Schnittstelle Typdeklarationen und als Hauptmerkmal die RT-Infostruktur, wie sie im folgenden beschrieben wird.

4.1.2 Die RT-Infostruktur

Anwendungen die TAO verwenden, müssen all ihre geplanten Ressourcenanforderungen spezifizieren. Diese Dienstgüteanforderung wird TAO vor der Programmausführung bereitgestellt. Im Falle des CPU-Schedulings werden die Dienstgüteanforderungen mit den folgenden Attributen der RT-Info-IDL-Struktur ausgedrückt.

- Worst-case execution time
  Ist die maximale Ausführungszeit der RT-Operation und wird für konservative Zeitanalysen mit strikten Echtzeitanforderungen verwendet
- Typical execution time
  Ist die Ausführungszeit, die eine RT-Operation für gewöhnlich benötigt und wird für statistische Echtzeitsysteme verwendet.
- Cached execution time
  Wenn eine Operation ein zwischengespeichertes Ergebnis zur Auftragsbearbeitung verwenden kann, wird die cached execution time auf einen Wert ungleich Null gesetzt. Bei der Ausführung periodischer Funktionen werden dann die worst-case Kosten nur einmal pro Periode fällig.
- Period
  Ist die minimale Zeit zwischen zwei aufeinanderfolgenden Operationen.
- Critically
  In diesem Zahlenwert wird festgelegt, wie kritisch die Operation ist. Der Wert ist wichtig für die Entscheidungen des Scheduling Service, wenn Operationen mit gleicher Priorität verarbeitet werden müssen.
- Importance
  Mit diesem Zahlenwert wird festgelegt, wie wichtig die entsprechende Operation ist. Dieser Wert wird vom Scheduling Service als Entscheidungshilfe herangezogen, wenn die Ausführung der RT-Operation nicht von Datenabhängigkeiten oder dem Critically Wert abhängt.
- Dependency Info
  Ist ein Array mit Einträgen über andere RT-Info Instanzen. Es gibt einen Eintrag für jede RT-Operation, von der diese abhängig ist. Diese Abhängigkeiten werden zur Scheduling

```
module RT_Scheduler
{
  // Module TimeBase defines the OMG Time Service.
  typedef TimeBase::TimeT Time;// 100 nanoseconds
  typedef Time Quantum;
  typedef long Period;// 100 nanoseconds

   enum Importance
  // Defines the importance of the operation,
  // which can be used by the Scheduler as a
  // "tie-breaker" when other scheduling
  // parameters are equal.

  {
    VERY_LOW_IMPORTANCE,
    LOW_IMPORTANCE,
    MEDIUM_IMPORTANCE,
    HIGH_IMPORTANCE,
    VERY_HIGH_IMPORTANCE
  };

  typedef long handle_t;
  // RT_Info's are assigned per-application
  // unique identifiers.

  struct Dependency_Info
  {
    long number_of_calls;
    handle_t rt_info;
    // Notice the reference to the RT_Info we
    // depend on.
  };

  typedef sequence<Dependency_Info> Dependency_Set;
  typedef long OS_Priority; typedef long Sub_Priority;
  typedef long Preemption_Priority;

  struct RT_Info
    // = TITLE
    // Describes the QoS for an "RT_Operation".
    // = DESCRIPTION
    // The CPU requirements and QoS for each
    // "entity" implementing an application
    // operation is described by the following
    // information.
  {
    // Application-defined string that uniquely
    // identifies the operation.
    string entry_point_;

    // The scheduler-defined unique identifier.
    handle_t handle_;

    // Execution times.
    Time worstcase_execution_time_;
    Time typical_execution_time_;

    // To account for server data caching.
    Time cached_execution_time_;

    // For rate-base operations, this expresses
    // the rate. 0 means "completely passive",
    // i.e., this operation only executes when
    // called.
    Period period_;

    // Operation importance, used to "break ties".
    Importance importance_;

    // For time-slicing (for BACKGROUND
    // operations only).
    Quantum quantum_;

    // The number of internal threads contained
    // by the operation.
    long threads_;

    // The following attributes are defined by
    // the Scheduler once the off-line schedule
    // is computed.

    // The operations we depend upon.
    Dependency_Set dependencies_;

    // The OS por processing the events generated
    // from this RT_Info.
    OS_Priority priority_;

    // For ordering RT_Info's with equal priority.
    Sub_Priority subpriority_;

    // The queue number for this RT_Info.
    Preemption_Priority preemption_priority_;
  };
};
```

Abbildung 8: TAOs RT-Scheduler[Doug98]

Analyse benützt, um Threads im System zu identifizieren. Jeder einzelne Abhängigkeitsgraph stellt einen Thread dar.

Die eben aufgeführten RIDL-Schemen können dazu benützt werden, um die Laufzeiteigenschaften von Objektoperationen für TAOs Scheduling Service zu bestimmen. Die Informationen werden von TAO einerseits dazu benützt, die Ausführbarkeit des Zeitplans zu bestimmen und andererseits für die Reservierung von ORB-Endsystemressourcen und Netzwerkressourcen für die Verarbeitung der RT-Operation. Jede RT-Operation benötigt eine RT-Info-Instanz.

5 Vergleich: Dynamisches vs. statisches CORBA

In diesem Kapitel soll kurz auf dynamische CORBA-Umgebungen eingegangen werden. Die Erfahrungen stammen aus einer Implementierung [Victa] der Universität von Rhode Island. Sie ist eine Erweiterung der CORBA-Implementierung ORBIX, der Firma Iona Technologies. Auch bei dynamischen CORBA-Umgebungen können Clients und Server sowohl hinzugefügt, als auch entfernt werden. Der wesentliche Unterschied zu den statischen Umgebungen besteht darin, dass während der Laufzeit die Dienstgüteanforderungen verändert werden können. Somit ist bei diesen Umgebungen eine a priori Analyse des gesamten Laufzeitverhaltens nicht möglich. Um die gegebenen Zeitschranken einzuhalten, wird nur die Strategie der "best-möglichen" Ausführung unterstützt. Für die Umsetzung dieser Strategie werden wesentliche Anforderungen an das zugrundeliegende Echzeitbetriebsystem gestellt, das dem POSIX-Standard entsprechen muss. Dennoch können im Gegensatz zu statischen Umgebungen keine harten Echtzeitgarantien gegeben werden.

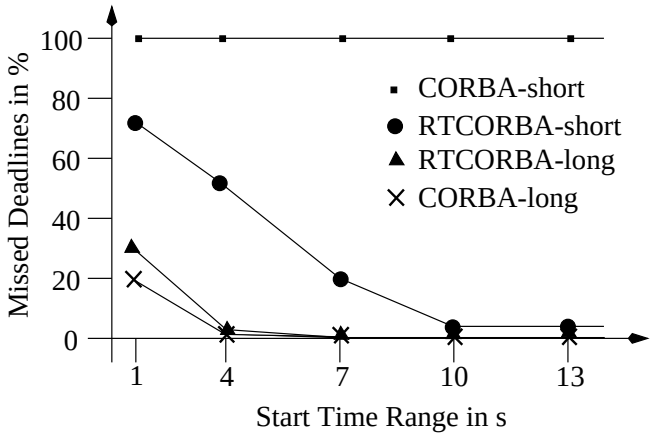


Abbildung 9: Deadlinetest der dynamischen CORBA-Umgebung von Rhode Island[Victa]

Die Messungen des Prototyps von Rhode Island wurden auf einem isolierten Netzwerk mit zwei Knoten und einer Netzverzögerung von 1,2ms gemacht. Die durchschnittliche Event Response Time lag im Bereich von 100ms und bei einem Deadlinetest wurden selbst Deadlines, die grösser als 3s waren nicht eingehalten (Vgl Bild9). Diese schwachen Ergebnisse ruhen von der langen Verarbeitungszeit der Dienstgüteparameter her.

Für die Messungen an TAO wurden zwei Dualprozessor UltraSPARC2 Rechner über ein ATM-Netzwerk verbunden. Abbildung 10 zeigt die guten Skalierbarkeitseingenschaften von TAO. Selbst bei hinzunahme von 50 Clients erhöht sich die Latenzzeit von 2ms, nur um etwa 0,7ms. Man erkennt auch, dass hochpriore Clients stets schneller verarbeitet werden als niederpriore. Dies belegt die Verwirklichung der Ende-zu-Ende-Prioritätsanforderungen. Insgesamt ist die statische Version von TAO (es gibt auch eine dynamische), mit einer Latenzzeit von 2ms um den Faktor 50 schneller als die dynamische Rhode Island Implementierung.

6 Abschliessende Bemerkungen

Die Entwicklung von Echtzeitsystemen geht zur Benutzung von Middlewarekomponenten von der Stange über, um die Kosten des Softwarelebenszyklus zu senken und die Zeit bis zur Vermarktung zu verkürzen. In diesem ökonomischen Umfeld ist das flexible und adaptierbare CORBA eine attraktive Middlewarearchitektur. Die Optimierungen und Leistungen, auf die in diesem Seminarbericht eingegangen wird, zeigen dass die CORBA ORBs der nächsten

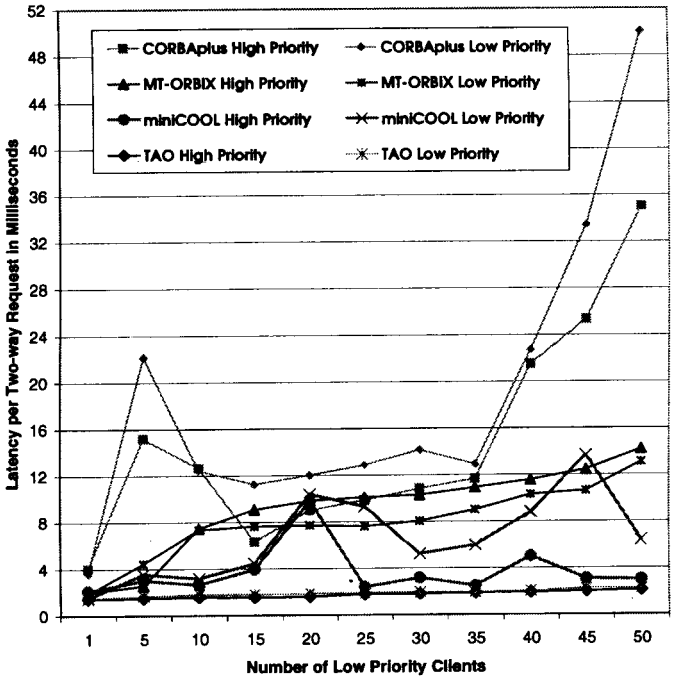


Abbildung 10: Vergleich der Latenzzeiten von CORBAplus, MT-Orbix, miniCOOL und TAO [Doug98]

Generation gut geeignet sein werden, für verteilte Echtzeitsysteme, die ihrerseits Effizienz, Skalierbarkeit und vorhersagbare Leistung fordern. Am Beispiel von TAO werden Optimierungen aufgezeigt, mit denen CORBA die Fähigkeit erlangt, harte Echtzeitanforderungen zu erfüllen.

Der C++ Quellcode für TAO ist im Internet unter der folgenden Adresse frei verfügbar,

```
http://www.cs.wustl.edu/~schmidt/TAO.html
```

Literatur

[Anir99a] Douglas C. Schmidt Aniruddha Gokhale. Measuring the Performance of Communication Middleware on High-Speed Networks. In *SIGCOMM '99*, Mai 1999, S. 306–317.

[Anir99b] Douglas C. Schmidt Aniruddha Gokhale, Irfan Pyarali. Design Considerations and Performance Optimizations for Real-time ORBs. In *5 th USENIX Conference on OO Technologies and Systems (COOTS '99)*, Mai 1999.

[Doug98] Chris Cleeland Douglas C. Schmidt, David L. Levine. Architectures and Patterns for Developing High-performance, Real-time ORB Endsystem. *Advances in Computers*, 1998.

[Mich98] Jishnu Mukerji Michel Ruffin, Judy McGoogan (Hrsg.). *Real-Time CORBA*. OMG TC orbos/98-12-10, 1998.

[Victa] Roman Ginis Victor Fay Wolfe, Lisa Cingiser DiPippo (Hrsg.). *Expressing and Enforcing Timing Constraints in a Dynamic Real-Time CORBA System.*

[Victb] Roman Ginis Victor Fay Wolfe, Lisa Cingiser DiPippo (Hrsg.). *Real-Time CORBA.*

A/V Streaming in CORBA

Robert Niess

Kurzfassung

Das Ziel dieser Ausarbeitung ist der Vergleich von herkömmlichen Lösungen zur Übertragung kontinuierlicher Daten mit den CORBA basierenden. Dazu werden RTSP (Real-Time Streaming Protocol), RTSS (Real-Time Streaming System for ATM Networks) und die OMG Spezifikation für A/V Streaming vorgestellt und miteinander verglichen.

1 Motivation

Durch die wachsende Bandbreite der Kommunikationsleitungen ist eine neue Klasse von Programmen entstanden, deren Kommunikation nicht mehr auf der Anfrage/Ergebnis Methode, sondern auf einem kontinuierlichen Datenstrom basiert. Als Beispiel wären da neben den Internet-Anwendungen RealVideo oder MP3-Streaming auch permanente Meßwerterfassung oder z.B. eine Operation, bei der sich der Patient mehrere km weit weg vom operierenden Arzt befindet, zu nennen. Viele Anwendungen benutzen für diese Aufgabe proprietäre Protokolle, was die Kommunikation zwischen den Produkten unterschiedlicher Hersteller behindert oder ganz unmöglich gestaltet. Durch den Einsatz von CORBA ist es möglich diese Anwendungen flexibel und portabel zu gestalten.

2 Anforderungen

Diese Art der Kommunikation hat natürlich Ihre eigenen Anforderungen. Zum einen sollte die Kommunikation möglichst ressourcenschonend funktionieren. Das gilt sowohl für die Belastung des Kommunikationsmediums als auch für die Belastung der Prozessoren auf der Client- und auf der Serverseite. Zusätzlich sollte eine Zusicherung über die erforderliche Bandbreite und eine bestimmte Übertragungszeit auf dem Kommunikationsmedium erfolgen (QoS, Quality of Service).

3 RTSP (Real Time Streaming Protocol)

Das Real Time Streaming Protocol ist ein von der IETF (Internet Engineering Task Force vorgeschlagener Standard für die Übertragung und Steuerung von multimedialen Daten über das Internet. Es findet unter anderem in der aktuellen Version des sehr beliebten RealPlayer G2 Verwendung. Das RTSP dient in erster Linie zur Steuerung und Synchronisierung von Streams. Es bietet Funktionen wie Abspielen, Pause, Vor- /Zurückspulen und eine absolute Positionierung innerhalb eines Streams, zusätzlich läßt es sich um eigene Steuerfunktionen erweitern. Für die eigentliche Übertragung der Daten wird üblicherweise RTP (Real Time Transport Protocol) benutzt, welches unter anderem QoS bietet. Das RTP läßt sich in das

RTSP einbetten, was den Umgang mit Firewalls erleichtert. Das RTSP ist in erster Linie für den Einsatz im Internet entwickelt worden, daher ist es stark an das HTTP Protokoll angelehnt:
Beispielanfrage (C - Client, M - Multimediaserver, A - Audioserver):

C -> M:
GET /twister HTTP/1.1
Host: www.content.com
Accept: application/sdf; application/sdp
M -> C:
200 OK
Content-type: application/sdf

(session (all
(media (t audio) (oneof (
(e PCMU/8000/1 89 DVI4/8000/1 90) (id lofi))
(e DVI4/16000/2 90 DVI4/16000/2 91) (id hifi))
) (language en) (id rtspu://audio.content.com/twister/audio.en))
(media (t video) (e JPEG) (id rtspu://video.content.com/twister/video))))
C -> A:
SETUP rtsp://audio.content.com/twister/audio.en/lofi RTSP/1.0 1
Transport: rtp/udp; compression; port=3056
A -> C:
RTSP/1.0 200 1 OK
Session:1234
C -> A:
PLAY rtsp://audio.content.com/twister/audio.en/lofi RTSP/1.0 2
Session: 1234
Range:smpte 0:10:00-
A -> C:
200 2 OK

Die Offenheit von RTSP ist ein großer Vorteil gegenüber proprietären Protokollen. Damit lassen sich Anwendungen unterschiedlicher Hersteller unter unterschiedlichen Betriebsystemen miteinander koppeln. Zu den weiteren Vorteilen gehören die freie Erweiterbarkeit der Steuerfunktionen und die freie Wahl des Transportprotokolls.

4 Motivation zum Einsatz von CORBA für A/V Streaming

Das Anfrage/Antwort Prinzip von CORBA scheint auf den ersten Blick für den Einsatz in Streaming Applikationen ungeeignet zu sein. Müßte man die zu übertragenen Daten Frame für Frame anfordern, würde das zu einer großen Verzögerung und großem Overhead auf dem Kommunikationsmedium führen. Andererseits würden die Applikationen von der Flexibilität und Portabilität von CORBA stark profitieren. Eine Lösung dieses Dilemmas besteht darin, daß man die Datenströme an CORBA vorbei führt und alles andere wie Verbindungsaufbau, Synchronisation und Steuerfunktionen mit CORBA durchführt. Folgende Abbildung soll das Prinzip verdeutlichen:

Jedes Ende eines Streams besteht aus drei logischen Einheiten:

- Stream Interface Control Object: Ein durch die IDL definiertes Interface zur Steuerung und Verwaltung des Streams

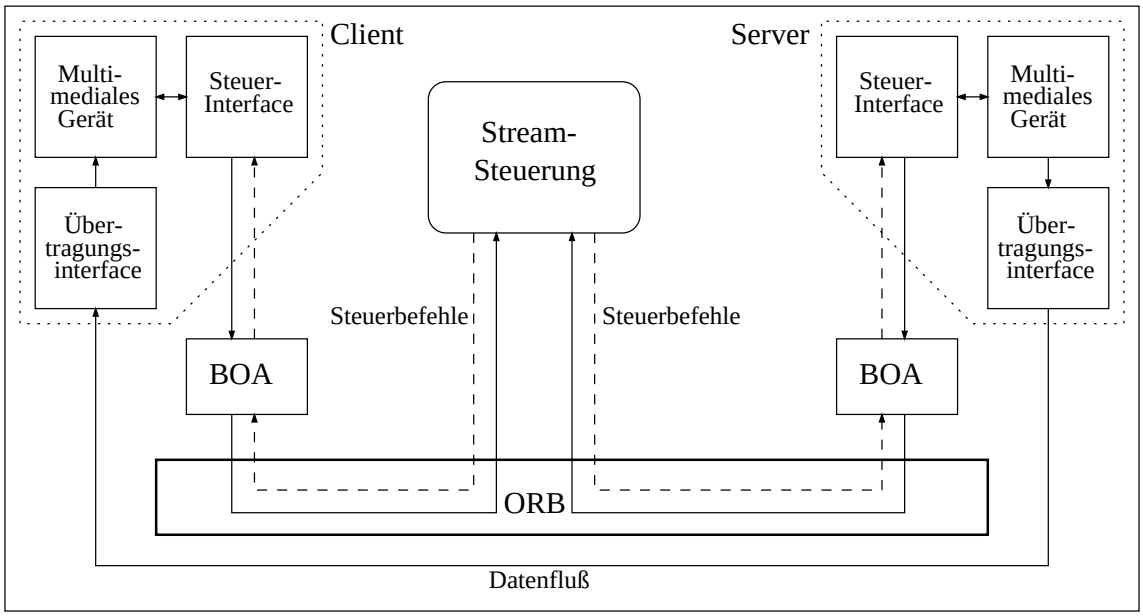


Abbildung 1: Schematischer Aufbau einer CORBA Lösung

- Flow data End-Point: Das Ziel oder Quelle eines Datenflußes
- Stream Adapter: Ist Zuständig für die Übermittlung des Datenflußes

Die Kommunikation zwischen dem Flow Data End-Piont, dem Stream Interface Control Object und dem Stream Adapter bleibt dem Anwendungsprogrammierer überlassen. Auf diesem Lösungsansatz basieren die OMG Spezifikation für A/V Streaming und der Real-Time Streaming Service für ATM Networks.

5 OMG Spezifikation für A/V Streaming

5.1 Einführung

Die CORBA A/V Streaming Spezifikation beschreibt ein Modell zur Implementierung eines multimedialen Streaming Systems, welches wohldefinierte Module, Schnittstellen und Semantik für den Verbindungsaufbau und Steuerung mit einem effizienten Datentransportprotokoll verbindet. Um sowohl einfachen Anwendungen mit einem unidirektionalen Stream, als auch komplexen Strukturen mit mehreren Streams und mehreren Erzeugern/Konsumenten, gerecht zu werden, hat man die Spezifikation in eine Light- und eine "Vollversion" aufgeteilt. Die "Vollversion" baut auf der "Lightversion" auf und ist abwärtskompatibel, d.h. es ist z.B. möglich zwischen einem Server, der die volle Spezifikation implementiert, und einem Client, der sich der einfacheren "Lightversion" bedient, eine Verbindung herzustellen. Dieses Model wurde bereits erfolgreich an der Washington University auf der Basis von TAO - einem echtzeit CORBA ORB - implementiert.

5.2 Aufbau und Funktionsweise

Durch die Spezifikation wird eine Reihe von Interfaces definiert, welche zusammen ein CORBA konformes Framework für die Arbeit mit Streams bilden. Im einzelnen sind das in der "Lightversion":

- MMDevice und VDev - reelle bzw. virtuelle Geräte
- StreamCrtl - Stream Steuerung
- SteramEndPoint - Endpunkt eines Streams

In der umfangreicheren "Vollversion" des Protokolls gibt es noch:

- FDev - Datenfluß Erzeuger oder Konsument
- FlowConnection und FlowEndPoint - Datenflüsse bzw. deren Endpunkte
- MediaControl - Umfangreichere Steuerungsschnittstelle

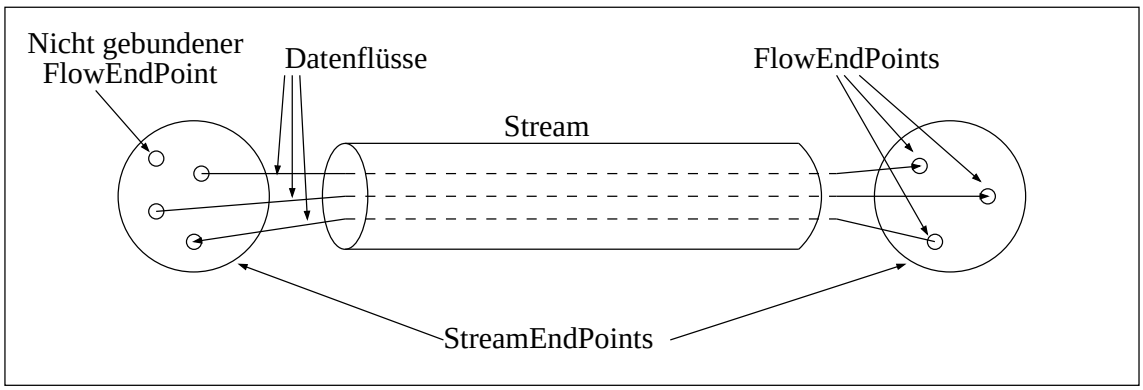


Abbildung 2: Aufbau eines Streams

Ein Stream bezeichnet eine Verbindung zwischen mehreren virtuellen Geräten, welche in der "Vollversion" aus mehreren Datenflüssen beliebiger Richtung bestehen kann. Ein Steuerbefehl auf einem Stream kann entweder alle oder auch nur eine Teilmenge der zugehörigen Datenflüsse steuern. Jeder Datenfluß stellt eine unidirektionale Verbindung dar, er hat immer eine Quelle und ein oder mehrere Ziele. Ein StreamEndPoint kann mehrere FlowEndPoints haben, die sowohl Quelle aus auch Ziel von Datenflüssen sein können. Nicht verbundene FlowEndPoints sind ebenfalls erlaubt.

5.2.1 MMDevice und VDev Interfaces

Das MMDevice Interface beschreibt ein multimediales Gerät, welches Daten erzeugt oder konsumiert. Das kann ein physikalisches Gerät wie ein Mikrofon, Lautsprecher, Bildschirm oder ein logisches Gerät wie eine Datenbank oder eine Datei auf der Festplatte sein. Es kann über ein oder mehrere Streams mit anderen MMDevices verbunden werden. Dazu wird für jede Verbindung ein VDev Interface und ein StreamEndPoint Interface erzeugt, in denen datenspezifische bzw. übertragungsspezifische Parameter festgehalten werden. Wenn zwei virtuelle Geräte miteinander verbunden werden, müssen sie sich auf eine identische Konfiguration einigen, dazu bieten sie die Funktionen set_format(), set_dev_params() und configure() an. Das Verhalten der Funktion, die das beste gemeinsame Format auswählen soll, wird dem Anwendungsprogrammierer überlassen. Das MMDevice bietet mit der Funktion bind() die Möglichkeit, es mit einem anderen MMDevice zu verbinden, das Ergebnis dieser Funktion ist ein StreamCtrl Interface, welches den neu erzeugten Stream zwischen den beiden MMDevices steuert.

5.2.2 StreamCtrl Interface

Das StreamCtrl Interface abstrahiert die kontinuierliche Datenübertragung zwischen virtuellen Geräten. Es bietet Funktionen zum Verbinden von MMDevices mittels eines Streams und die grundsätzlichen Steuerbefehle wie Start und Stop. Das Interface läßt sich um weitere Steuerbefehle, wie z.B. Zurückspulen oder absolute Positionierung innerhalb des Streams, bei Bedarf erweitern.

Die vorher erwähnte bind() Funktion des MMDevice ist eine Abkürzung für das Erzeugen eines StreamCtrl und den darauf folgenden Aufruf von bind_devs(). Das StreamCtrl Interface bietet zusätzlich die Möglichkeit Multicast Streams aufzubauen, bei denen das sendende MMDevice identische Daten an mehrere Empfänger sendet. Das kann bei entsprechender Unterstützung des Übertragungsmediums die Netzlast erheblich reduzieren.

5.2.3 StreamEndPoint Interface

Die OMG Spezifikation definiert 2 Typen von Endpunkten eines Streams: 1) A-Typ - StreamEndPoint_A und 2) B-Typ - StreamEndPoint_B. Beide können sowohl das erzeugende als auch das konsumierende Ende des Streams darstellen, aber ein StreamEndPoint_A kann nur mit einem StreamEndPoint_B verbunden werden. Dadurch wird sichergestellt daß zwei Erzeuger oder zwei Konsumenten nicht miteinander verbunden werden und es wird damit gleichzeitig die Richtung des Datenflusses festgelegt. Für ganz einfache Streaming Aufgaben ist es sogar möglich die MMDevices und VDevs wegzulassen und eine Verbindung zwischen zwei StreamEndPoints über ein StreamCtrl Objekt mit der Funktion bind() direkt herzustellen. Dabei wird die Möglichkeit von Multicast Streams ebenfalls unterstützt.

5.2.4 Quality of Service (QoS)

Die Funktionen bind() und bind_devs() erlauben die Angabe von diversen QoS- Parametern, die ebenfalls in dieser Spezifikation festgelegt sind. In der Regel beschreiben diese QoS-Parameter die Anforderungen an die Daten, wie z.B. die Framerate, Videogröße oder die Samplingfrequenz von Audiodaten, die Umrechnung auf die netzwerkspezifischen QoS Parameter, wie Bandbreite oder maximale Verzögerung, erfolgt erst nach der Auswahl des Transportprotokolls. Eine nachträgliche Änderung der QoS Parameter kann mit der Funktion modify_QoS() jederzeit durchgeführt werden.

5.2.5 Verbindungsablauf bei der "Lightversion" der Spezifikation

1. Sobald der Anwendungsprogrammierer das lokale MMDevice initialisiert und ein entferntes MMDevice z.B. über den Trading Service ausfindig gemacht hat wird ein StreamCtrl Objekt erzeugt und die Funktion bind_devs() aufgerufen.

2. Das StreamCtrl Objekt weist beide MMDevices durch den Aufruf der Funktionen create_A() bzw. create_B() an je ein VDev und ein StreamEndPoint Interface zu erzeugen. Sollte ein MMDevice keine weiteren Verbindungen mehr akzeptieren oder z.B. nur den StreamEndPoint_A anbieten, dann wird eine streamOpDenied Exception geworfen.

3. Sind beide Aufrufe erfolgreich gewesen, dann wird jedem VDev mit dem Aufruf der Funktion set_peer() das jeweils andere VDev bekanntgegeben.

4. In dem set_peer() Aufruf konfigurieren sich die VDevs gegenseitig mit den Funktionen set_format(), set_dev_params() und configure(), dabei wird das Übertragungsformat der Daten festgelegt.

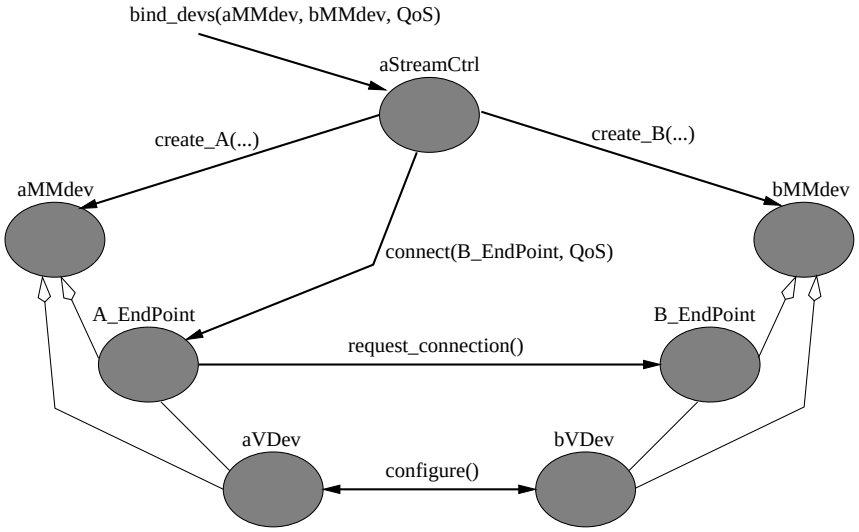


Abbildung 3: Verbindungsaufbau

5. Jetzt kehrt die Funktion bind_devs() zurück und beide Seiten sind für den Aufbau des Streams bereit. Das geschieht durch den Aufruf von connect() auf einem von den zwei StreamEndPoints.

6. Dieser StreamEndPoint ruft daraufhin die Funktion request_connection() des anderen StreamEndPoints auf, welcher daraufhin die Verbindung aufbaut.

5.2.6 Bidirektionale Streams

In der "Lightversion" der Spezifikation kann man nur unidirektionale Streams aufbauen, möchte man allerdings komplexere Anwendungen, wie z.B. eine Videokonferenz Anwendung entwickeln, dann muß man die "Vollversion" der Spezifikation implementieren. Sie ermöglicht es mehrere Datenflüsse beliebiger Richtung in einem Stream unterzubringen und sie gemeinsam - oder auch einzeln - zu steuern. Dazu sind werden weitere Interfaces definiert:

5.2.7 FDev Interface

Was MMDevice für einen Stream ist, ist das FDev für einen Datenfluß, im Unterschied zu dem MMDevice erzeugt das FDev allerdings nur einen FlowEndPoint, das Pendant zum VDev ist in das FlowEndPoint Interface bereits integriert.

5.2.8 FlowEndPoint Interface

Das FlowEndPoint Interface entspricht in der Funktionsweise dem StreamEndPoint Interface, es ist für die Steuerung von Datenflüssen zuständig. Ähnlich zu den A- und B-Typen der StreamEndPoints existieren zwei Varianten der FlowEndPoints: FlowProducer und FlowConsumer. Zusätzlich bieten sie die Funktion is_fep_compatible() an, mit der man die Kompatibilität zweier FlowEndPoints überprüfen kann. Die Kompatibilität ist gegeben, wenn beide FlowEndPoints ein gemeinsames Übertragungsprotokoll und ein gemeinsames Datenformat unterstützen. Mehrere FlowEndPoints können mit der Funktion set_fep() einem StreamEndPoint zugeordnet werden.

5.2.9 FlowConnection Interface

Dieses Interface repräsentiert den Datenfluß zwischen zwei FlowEndPoints, es bietet Funktionen zum Auf-/Abbau der Verbindung sowie zur Steuerung des Datenflusses. Mehrere FlowConnections können mit der Funktion set_flow_connection() einem StreamCtrl zugeordnet werden.

5.2.10 MediaControl Interface

Dieses Interface bietet umfangreichere Befehle für die Steuerung eines Streams oder eines Datenflusses. Es dient der Erweiterung des StreamCtrl oder des FlowConnecton Inferface.

5.2.11 Verbindungsaufbau bei der "Vollversion" der Spezifikation

Wird die Funktion bind() oder bind_devs() auf einem StreamCtrl Objekt aufgerufen, dann fordert es von jedem StreamEndPoint die Liste der zugehörigen FlowEndPoints an. Sollte einer oder beide StreamEndPoints keine FlowEndPoints besitzen, dann wird für die Verbindung die "Lightversion" benutzt, ansonsten findet eine Kompatibilitätsüberprüfung der StreamEndPoints statt. Sie sind kompatibel, wenn es mindestens ein kompatibles Paar von FlowEndPoints existiert. Anschließend wird für jedes kompatible Paar je ein FlowConnection Objekt erzeugt und darauf die Funktion connect() aufgerufen, welche ihrerseits auf dem FlowConsumer die Funktion go_to_listen() und auf dem FlowProducer die Funktion connect_to_peer() aufruft.

5.2.12 SFP - Simple Flow Protocol

Die OMG Spezifikation definiert mit dem SFP ein einfaches Transportprotokoll welches die einheitliche Übertragung der Daten unabhängig von den verschiedenen Netzwerkprotokollen ermöglicht. Außerdem werden die Daten dabei mit einem Zeitstempel versehen, welcher die spätere Synchronisierung der Daten erlaubt. Die Benutzung des SFP wird von der OMG nicht vorgeschrieben, die einzelnen FlowEndPoints können selber entscheiden ob es eingesetzt werden soll oder nicht.

5.3 Beispiel Implementierung

Wenn man nun eine verteilte Anwendung nach dieser Spezifikation schreiben wollte, wie würde man da vorgehen? Als erstes kreiert man eine Reihe von Klassen, welche auf den jeweiligen IDL Interfaces basieren. Die Namenskonvention ist dabei durch die Spezifikation vorgeschrieben:

- X_StreamCrtl - abgeleitet von StreamCtrl
- X_A und X_B - abgeleitet von StreamEndPoint_A bzw. StreamEndPoint_B
- X - abgeleitet von MMDevice
- v_X - abgeleitet von VDev

für die Datenflüsse:

- Y_FlowConnection - abgeleitet von FlowConnection

- Y_Consumer und Y_Producer - abgeleitet von FlowConsumer bzw. FlowProducer
- F_Y - abgeleitet von FDev

X_StreamCtrl und Y_FlowConnection können durch Mehrfachvererbung zusätzlich von dem MediaControl Interface abgeleitet werden um mehr Kontrolle über den Stream bzw. Datenfluß zu erhalten.

Dann erweitert man diese Klassen um die für die Anwendung notwendige Funktionen, wie die Erzeugung oder Verarbeitung der Daten. Die Implementierung eines Videokonferenzsystems könnte dann z.B. aussehen wie in Abbildung 4.

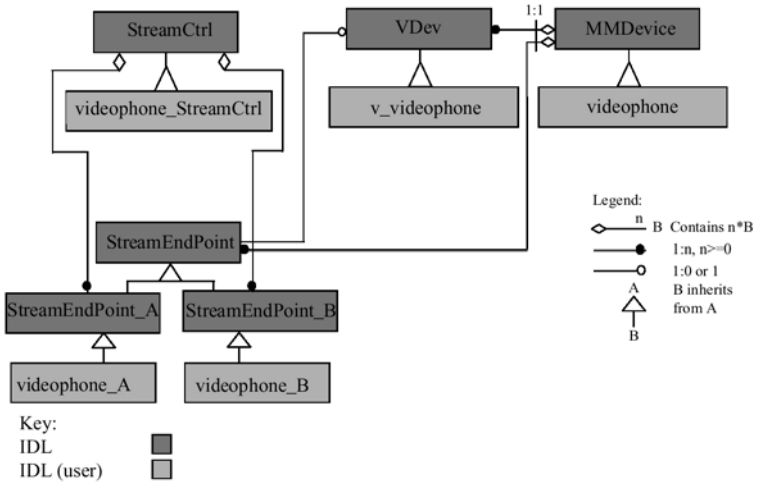


Abbildung 4: Beispielanwendung: Videokonferenzsystems

Nach diesem Modell wurde an der Washington University ein Video Server und ein entsprechender Client entwickelt. Dabei wurden zwei Streams benutzt - einer für Video (MPEG Format), und einer für Audio (Sun ULAW Format). Als Übertragungsprotokoll fand dabei UDP den Einsatz. Der Server bietet die Steuerfunktionen play(), stop() und rewind() an. Die zu übertragenen Daten lagen als Dateien auf der Festplatte vor. Als Client hat man einen bestehenden MPEG Player genommen und ihn um die Streaming Funktionen erweitert.

5.4 Performance

Mit der Implementierung der Washington University wurden ein paar Benchmarks durchgeführt, das Ergebnis eines davon möchte ich hier Vorstellen:

Hier wurde der maximale Datendurchsatz auf einer ATM Strecke gemessen. Dabei hat man das vorgestellte Streaming Modell mit einem Octet Stream, wie er von CORBA bereitgestellt wird, verglichen. Zusätzlich hat man den theoretischen Maximaldurchsatz in das Diagramm aufgenommen. Wie man sieht, liegt das Ergebnis des vorgestellten Streaming Modells ziemlich nah an dem theoretischen Maximaldurchsatz, der Octet Stream kann jedoch erst bei einer großen Puffergröße einen angemessenen Datendurchsatz erreichen. Außerdem wird bei dem Octet Stream die CPU viel stärker belastet, da die Daten jedesmal durch den ORB wandern und dabei evtl. mehrfach kopiert werden müssen.

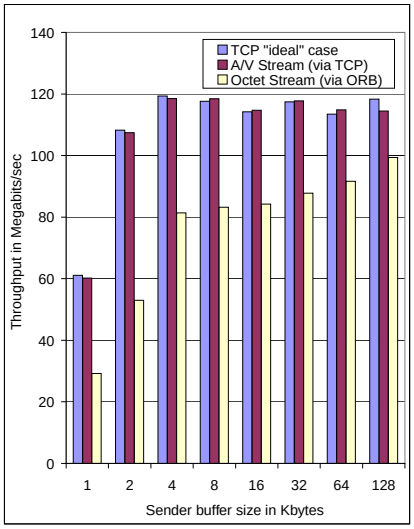


Abbildung 5: Performance

6 Real-Time Stream Service for ATM Networks

6.1 Einführung

Der Real-Time Stream Service for ATM Networks wurde an der National University of Singapure entwickelt. Das Ziel dabei war die Bereitstellung von objektorientierten, programiersprachen- und plattformunabhängigen Interfaces für die schnelle Entwicklung verteilter Anwendungen - was den Einsatz von CORBA nahelegte. Es wurde vor allem Wert auf die Unterstützung unterschiedlicher Transportprotokolle, QoS-Anforderungen und Echtzeitfähigkeit gelegt.

6.2 Aufbau

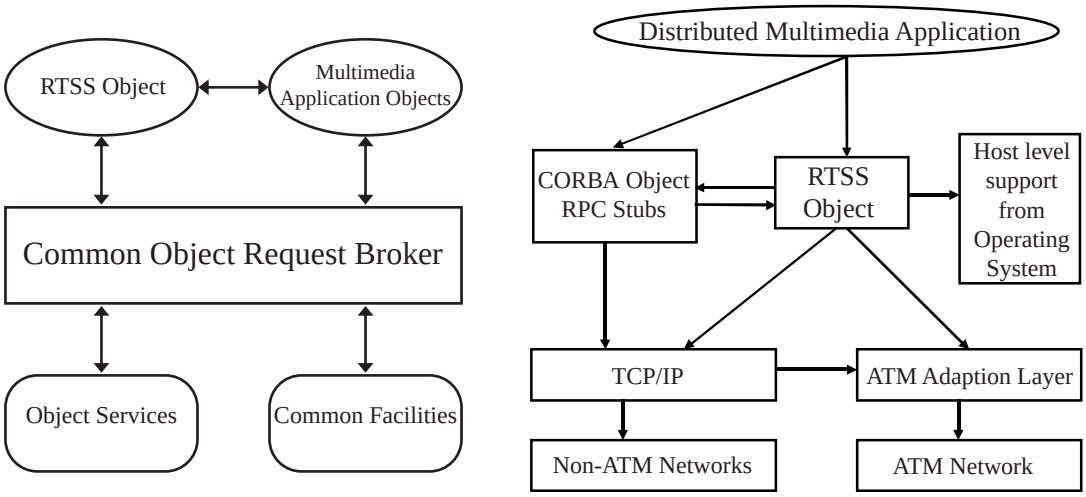


Abbildung 6: Schematischer Aufbau

Die Kommunikation zwischen dem RTSS Objekt und anderen Objekten erfolgt durch vorgegebene IDL Interfaces. Eine multimediale Anwendung, die sich im gleichen Adressraum mit dem RTSS Objekt befindet, kann jedoch direkt mit dem RTSS Objekt kommunizieren, dies

wird für die Anlieferung oder Abholung der zu übertragenen Daten oder für die Behandlung von Verletzungen der QoS Anforderungen benutzt. Um die Echtzeitanforderungen zu erfüllen greift das RTSS Objekt direkt auf die Bibliotheken des zugrundeliegenden Betriebssystems zu. Der interne Aufbau von RTSS ist in der Abbildung 7 dargestellt.

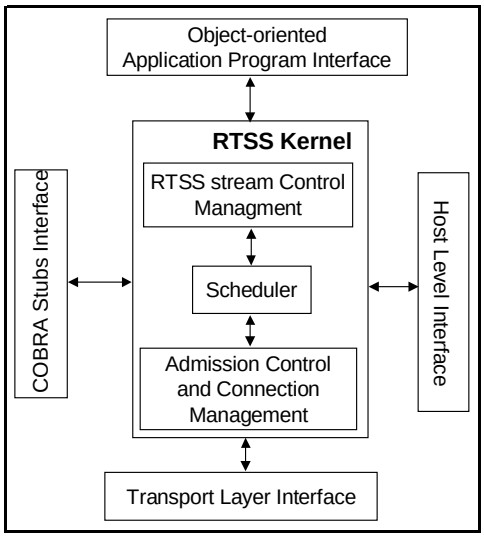


Abbildung 7: Interner Aufbau

6.2.1 Das objektorientierte API

Es stellt die direkte Verbindung zu der multimedialen Anwendung dar. Durch dieses API werden dem RTSS Objekt Behandlungsroutinen für die Erzeugung/Verarbeitung der Daten sowie Behandlungsroutinen für QoS Verletzungen bereitgestellt. Auf dieses API kann nur von dem Host, auf dem sich auch das RTSS Objekt befindet, zugegriffen werden.

6.2.2 CORBA Stubs Interface

Dies ist die Schnittstelle, mit der das RTSS mit der Außenwelt kommunizieren kann.

6.2.3 Transport Layer Interface

Dieses Interface stellt die Verbindung für den Austausch der Daten dar. Momentan werden dabei TCP/IP, UDP/IP und ATM unterstützt, wobei pro RTSS Objekt nur ein Transportprotokoll eingesetzt werden kann.

6.2.4 Host Level Interfaces

Sie stellen die Schnittstelle zum Betriebssystem dar. Durch sie werden Threads für die Verwaltung der Daten erzeugt und verwaltet. Dieser direkte Eingriff ist bei der Echtzeitanforderung an RTSS zwingend notwendig.

6.2.5 Scheduler

Der Scheduler ist für die Überwachung, Steuerung und Verwaltung aller interner Aktivitäten des RTSS zuständig. Er verteilt die Prioritäten der einzelnen Threads und kümmert sich um die Einhaltung der QoS Anforderungen jedes einzelnen RTSS Objekts.

6.2.6 Admission Control und Connection Manager

Dieses Modul ist für die Verwaltung der Ressourcen auf dem Host zuständig, dazu gehört der Speicher für die Buffer und auch die CPU Rechenleistung. Es übersetzt die anwendungsspezifische in netzwerkspezifische QoS Anforderungen, alloziert die dafür notwendigen Ressorcen und baut die Verbindung auf, sofern die QoS Anforderungen erfüllbar sind.

6.2.7 RTSS Stream Control Manager

Diese Modul bieten die Steuerfunktionen für einen Stream an, das sind im einzelnen start, stop, pause und resume.

6.3 Funktionsweise

Jeder Knoten eines verteilten System, welcher das RTSS bietet, hat eine RTSS Factory, welche für das Erzeugen und Verwalten von RTSS Ports zuständig ist. Der Programmierer kann mit dem Aufruf der Funktion RTSS::create_port (RTSS_name, RTSS_type, RTSS_qos) das Erzeugen eine RTSS Ports auf dem durch RTSS_name angegebenen Knoten veranlassen. RTSS_name ist ein eindeutiger Name eines Ports und hat die Form "object_name:server_name:host_name". RTSS_type ist entweder RTSS_read oder RTSS_write. RTSS_qos ist eine Sturktur, welche u.a. die Angaben zum Netzwerkprotokoll, Puffergröße, die Rate in Puffer/Sek., mit der der Puffer übertragen werden soll, und die CPU-Zeit in Millisekunden, die für das Abholen oder Anliefern der Daten in den Buffer benötigt wird. Eine Referenz auf einen bereits erzeugten Port kann mit RTSS::get_port (RTSS_name) geholt werden, und mit RTSS::delete_port (RTSS_port) kann ein Port entfernt werden.

Hat man einen Port erzeugt, muß man für die Anlieferung/Abholung der Daten sorgen. Dazu wird mit RTSS::attach_handler (RTSS_port, RTSS_data_handler) eine Funktion dem Port zugewiesen, welche von ihm aufgerufen wird, sobald der Puffer neue Daten benötigt oder die Daten an die Anwendung abgeben will. Diese Funktion bekommt als Parameter den Zeiger auf den Puffer und seine Größe. Da diese Funktion direkt auf den Puffer zugreifen soll, muß sie sich im gleichen Adressraum, wie der RTSS Port befinden. Mit RTSS::attach_qos_handler (RTSS_port, RTSS_qos_handler) kann man zusätzlich eine Funktion für die Behandlung von Verletzungen der QoS Anforderungen angeben, sie wird aufgerufen, wenn z.B. ein Teil der Daten nicht oder zu spät angekommen ist. In dieser Funktion kann der Programmierer entscheiden ob er den Stream weiter aufrechterhalten will.

Mit RTSS::connect (RTSS_port, RTSS_port) werden zwei Ports miteinander Verbunden. Einer der Ports muß vom Typ RTSS_read, der andere RTSS_write sein. Die Verbindung ist eine unidirektionale Punkt-zu-Punkt Verbindung, Eine bidirektionale Verbindung kann durch Erzeugen mehrerer RTSS Ports hergestellt werden, es fehlt dann aber eine gemeinsame Steuermöglichkeit. Multicast Verbindungen (ein Sender, mehrere Empfänger) ist noch nicht implementiert.

Zur Steuerung eines Streams werden vom RTSS Port die Funktionen start(), stop(), pause(), resume() angeboten, dabei wechselt der Status des Streams zw. connected, life und paused. Diese Funktionen können auf jedem der zwei, an der Verbindung beteiligten, Ports aufgerufen werden.

Je nach dem Status des Streams, können nur bestimmte Funktionen aufgerufen werden, mit Ausnahme von delete(), welche jederzeit den Port zerstört. Mit den Funktionen get_position(), get_status(), get_QOS() und get_error() kann der Status der Verbindung abgefragt werden.

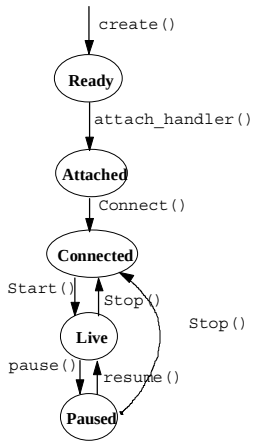


Abbildung 8: Statuswechsel Diagramm

Schließlich gibt es noch die Funktionen RTSS::export() und RTSS::release(), mit denen die RTSS Factory in einem Netzwerk für die Außenwelt bereitgestellt bzw. wieder entzogen wird. Diese werden normalerweise sofort nach dem Start bzw. kurz vor dem beenden des Servers aufgerufen.

6.4 Beispiel-Implementierung

Die Abbildung 9 demonstriert die Benutzung des RTSS in einer einfachen Anwendung:

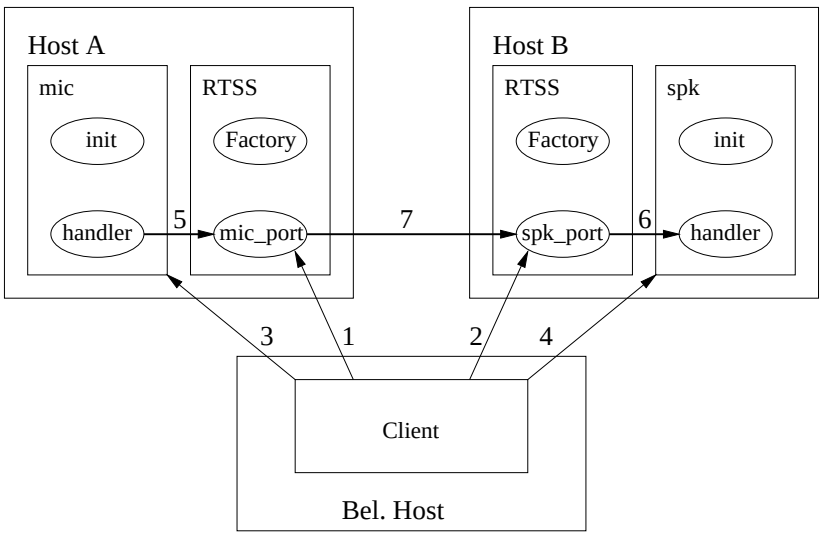


Abbildung 9: Beispielanwendung

Der Client erzeugt je ein RTSS_port für das Mikrofon und den Lautsprecher (1,2). Dann holt er Referenzen auf das Mikrofon Objekt und das Lautsprecher Objekt (3,4) und weist sie an, Daten- und QoS Behandlungsroutinen an die RTSS_ports zu binden (5,6). Anschließend werden die Ports durch den Aufruf der Funktion RTSS::connect() miteinander verbunden (7). Nun kann der Client den Stream mit den Funktionen start(), stop(), pause() und resume() steuern. Pseudocode:

main ()
1. mic_port = RTSS::create_port (<mic-name>, RT_WRITE, audio_QOS);
2. spk_port = RTSS::create_port (<spk-name>, RT_READ, audio_QOS);

3. mic = microphone::_bind();
4. spk = speaker::_bind();
5. mic->attach_rtss(<spk-name>);
6. spk->attach_rtss(<mic-name>);
7. RTSS::connect(mic_port, spk_port);
8. mic_port->start();
9. spk_port->start();
10. .....

6.5 Performance

Auch hier wurde der Datendurchsatz auf einer ATM Strecke gemessen:

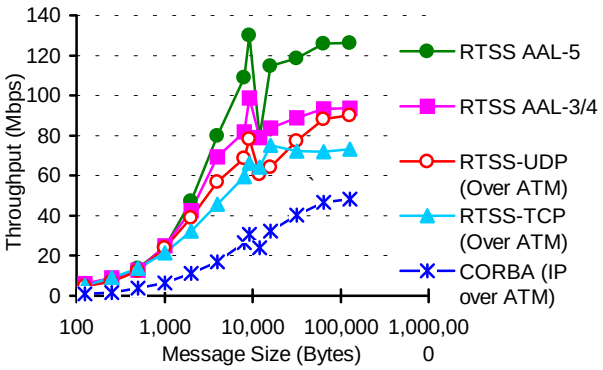


Abbildung 10: Performance

Auch hier zeigt es sich, daß mit zunehmender Puffergröße der Durchsatz bei dem Octet Stream von CORBA zwar zunimmt, aber trotzdem weit hinter den Ergebnissen vom RTSS liegen.

7 Vergleich und Schlußfolgerung

7.1 RTSP

Der größte Vorteil des RTSP ist seine Einfachheit. Es ist optimal für die Nutzung im Internet zusammen mit Web Browsern ausgelegt. Weder der Client noch der Server ist auf irgendwelche fremden Komponenten im System angewiesen.

7.2 OMG Spezifikation

Die OMG Lösung ist von allen die flexibelste. Durch die Unterteilung in "Voll-" und "Light-version" kann sie sowohl einfachen als auch komplexen Client/Server Architekturen genügen. Da es nur eine Spezifikation ist, bleibt dem Programmierer allerdings viel Arbeit überlassen, welche allerdings mit größtmöglicher Flexibilität und Interoperabilität belohnt wird.

7.3 RTSS for ATM Networks

Bei der Entwicklung des RTSS wurde u.a. Wert auf die Echtzeitfähigkeit und die Entlastung des Programmierers gelegt. Es genügt 2 Ports zu erzeugen, Behandlungsroutinen für Daten

und QoS anzugeben und die Ports miteinander zu verbinden. Der Preis dafür ist die geringere Flexibilität als bei der OMG Spezifikation, da nur unidirektionale point-to-point Streams unterstützt werden. Außerdem muß sowohl bei dem Client als auch bei dem Server neben dem ORB auch das RTSS Modul installiert sein - das macht es für den breiten Einsatz im Internet ungeeignet.

7.4 Schlußfolgerung

Beide CORBA basierte Lösungen zeichnen sich durch Flexibilität und Portabilität gegenüber dem RTSP aus. Der Einsatz von RTSS ist vor allem in der Industrie zu sehen, wo höchste Anforderungen an die Einhaltung von QoS gestellt werden. Applikationen, die nach der OMG Spezifikation entwickelt wurden, kann man überall einsetzen, wo ein ORB zur Verfügung steht. Und sobald die meisten Systeme über einen ORB verfügen werden, wird dem breiten Einsatz im Internet nichts mehr im Wege stehen - die Applikationen würden von dermaßen hohen Flexibilität nur profitieren.

7.5 Anmerkung

Während die A/V Streaming Spezifikation von OMG im TAO aktiv weiter gepflegt wird, konnte ich über den aktuellen Status von RTSS, welches 1996 entwickelt wurde, nichts, außer ein Paar "broken links", in Erfahrung bringen.

Literatur

[MaSS98] Sumedh Mangee, Nagarajan Surendran und Douglas C. Schmidt. *The Design and Performance of a CORBA A/V Streaming Service*. 1998.

[SPNW96] Bhawani S. Sapkota, Hung Keng Pung, Lek Heng Ngoh und Lawrence Wong. *RTSS: A CORBA-Based Real-Time Stream Service for ATM Networks*. National University of Singapure. 1996.

[Tele98] CORBA Telecoms. *Audio/Video Strams v1.0*. OMG. 1998.

Jini – Java Intelligent Network Infrastructure

Urs Jetter

Kurzfassung

Sun hat mit Jini ein Konzept vorgestellt, das die Bereitstellung von Diensten und Geräten in einem Netzwerk beinahe administrationsfrei regelt. Dieser Text handelt von den Grundideen hinter der Technologie. Daneben werden auch die Rechte und Pflichten, die sich aus der neuen Sun Community Source License ergeben, beleuchtet.

1 Einleitung

Nach der spektakulären Einführung von JAVA hat Sun jetzt mit Jini ein Konzept vorgelegt, das die Kommunikation von Endgeräten und Diensten innerhalb eines Netzwerks standardisieren soll. Jini bedeutet Java Intelligent Network Infrastructure. Dabei treten relativ wenig Einschränkungen oder Voraussetzungen auf:

Netzwerk: Alle Endgeräte und Dienste müssen einen dem Hersteller/Programmierer überlassenen Zugang zum Netzwerk besitzen und innerhalb dieses Netzwerks ein wahlfreies aber gleiches Protokoll sprechen.

Java Virtual Machine (JVM): Jeder Teilnehmer muß Zugang zu einer Java Virtual Machine haben.

Lookup Service: Mindestens ein Nachschlagdienst (Jini Lookup Service) muß installiert und erreichbar sein.

Trotz dieser wenigen Einschränkungen ist vor allem die für den Dienst oder das Endgerät notwendige Java Virtual Machine eine Voraussetzung, die hohe Kosten verursachen kann. Sun bietet zu dieser Thematik eher konservative Vorschläge an.[Micr99f] (siehe Abschnitt 3)

Bemerkenswert ist auch, daß Sun den Jini-Quellcode veröffentlicht hat. Damit soll die Weiterentwicklung und Anpassung erleichtert werden. Der Sourcecode und eine Menge Papier in Form von Spezifikationen unterliegen der neuen Sun Community Source License. Welche Rechte und Pflichten sich daraus ergeben, behandelt Abschnitt 5.

2 Aufbau von Jini-Netzwerken

Die Jini-Technologie ist unabhängig vom verwendeten Netzwerk. Das bedeutet nicht, daß die Technik eine Kommunikation zwischen Telefonen am Telefonnetz und Lampen am Stromnetz ermöglicht. Vielmehr baut Jini auf die Java Virtual Machine und die dort zugrundliegenden Kommunikationsmöglichkeiten des jeweiligen Betriebssystems und der Hardware.

Ein Jini-Service oder Jini-Gerät muß also – irgendwie – mit einem Jini-Servicenutzer kommunizieren können. Im Internet bedeutet das, daß beide Seiten über das TCP/IP-Protokoll miteinander in Verbindung treten können.

Trotzdem ist die Kommunikation keine reine Zweipunktverbindung. Während des Verbindungsaufbaus wird ein Lookup-Server befragt, ob und von wem denn ein geeigneter Service angeboten wird.

Neben des Lookup-Service ist auch Transaktionssicherheit ein Teil der Jini-Spezifikation. Kurz nach der Einführung der entsprechenden Objektrahmen hört die Spezifikation auch wieder auf: Jeder Teilnehmer muß selbst für geeignete Recoverymechanismen sorgen.

Wie bei anderen Systemen üblich, gilt eine Transaktion erst als abgeschlossen, wenn alle Teiltransaktionen korrekt abgeschlossen wurden. Andernfalls sollen die Änderungen verworfen werden.

Unter verteilten Ereignissen (distributed events) wird im Jini-Umfeld die Benachrichtigung von anderen Teilnehmern verstanden. In Erweiterung des von JavaBeans bekannten Systems schließen sich verschiedene Services zusammen, um bestimmte events gemeinsam zu bearbeiten. Dabei kann ein event auf dem einen Rechner eine Objekt auf einem anderen Rechner aufrufen.

Zunächst werden in diesem Kapitel die Teilnehmer mit Ihren Funktionen, und anschließend ihr Zusammenspiel im Jini-Netzwerk beschrieben.

2.1 Die Jini-Teilnehmer

Ein Lookup-Service ist ein Server, der permanent auf Anfragen von Dienstanbietern oder Dienstnehmenr wartet, um die Ressourcen der Dienstanbieter zu verteilen. Dabei hat er folgende funktionalität:

Lookup-Tabelle: Datenbank, die Informationen über Dienstanbieter und Zugangsbeschränkungen enthält

Discovery: Antwort auf alle Suchanfragen per Broadcast

Join: Einschreiben von Serviceanbietern in die Datenbank

Lookup: Auffinden eines geeigneten Service

Leaseing: Vergabe einer Ressource an einen Servicenehmer für eine bestimmte Zeit

Unter Dienstanbietern (Service Provider, Service) versteht Jini eine möglichst wenig eingeschränkte Menge: Endgeräte, Programme und Programmteile können einen Dienst anbieten. Ein Dienstanbieter beherrscht folgende Funktionen:

Discovery: Aufsuchen von Lookup-Services mit Hilfe eines Broadcasts

Join: Einschreiben bei einem Lookup-Service

Service: Der Dienst selbst

Servicenehmer (Service Client, Client) sind im Jini-Umfeld alle Applikationen, die einen Dienstanbieter inanspruchnehmen. Sie haben folgende Funktionen:

Discovery: Aufsuchen von Lookup-Services mit Hilfe eines Braodcasts

Lookup: Veranlaßt den Lookup-Service eine Menge von möglichen Serviceprovidern anzubieten

Leasing: Buchen eines Service

code injection: Ausführen eines Programmstückes, das der Lookup-Service zur Nutzung des Dienstanbieters bereitstellt

2.2 Anmeldung am Netzwerk – Discovery and Join

2.2.1 Discovery

Die Lookup-Technologe basiert auf der Überlegung, daß Ressourcen zentral verwaltet und nachgefragt werden können. Als Bild wird oft die Börse oder der Markt - noch entkoppelt von der monetären Komponente - herangezogen. Der Discovery and Join Mechanismus erlaubt es den Geräten einen Marktplatz zu finden, dort Services anzubieten oder auch nachzufragen.

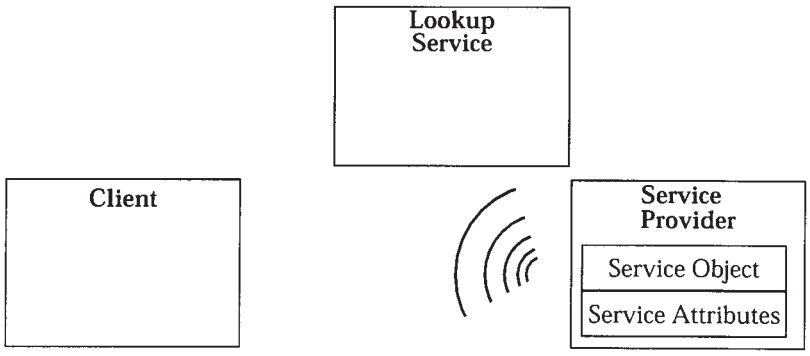


Abbildung 1: Discovery

Ein Service oder Gerät sucht beim Einschalten mittels eines Broadcast nach allen im System vorhadenen Lookup-Services. Es gibt belibig viele Lookup-Services im Netzwerk, so daß die Antwort sehr groß sein kann. Da die meisten Gateways Broadcasts nicht weiterrouten meint man damit der Netzwerksicherheit genügen zu können. Hier verläßt man sich allerdings auf Verhalten der Netzinfrastruktur, das nicht immer gesichert sein muß.

Alle Lookup-Services, die das Broadcast-Paket erhalten haben, senden jetzt ihre Antwort. Bei jedem Lookup-Service meldet sich das Gerät an. Man kann sich vorstellen, daß die Existenz vom mehr als einem Lookup-Service im Netz überflüssig ist - trotzdem ist sie möglich.

2.2.2 Join

Jeder Serviceanbieter übergibt mindestens ein Paar bestehend aus Serviceobject und Serviceattributen an jeden ihm bekannt gewordenen Lookup-Service. Ein Serviceanbieter mit mehreren belegbaren parallelen Ressourcen wird jede Ressource einzeln anmelden, damit sie getrennt zugeteilt werden können (z.B. ISDN-Kanäle).

Unabhängig von seinen Eigenschaften ist aber jeder Service, der unter Jini nutzbar sein soll im Lookup-Service registriert. Dazu übergibt der Service beim Anmelden folgende Serviceattribute:

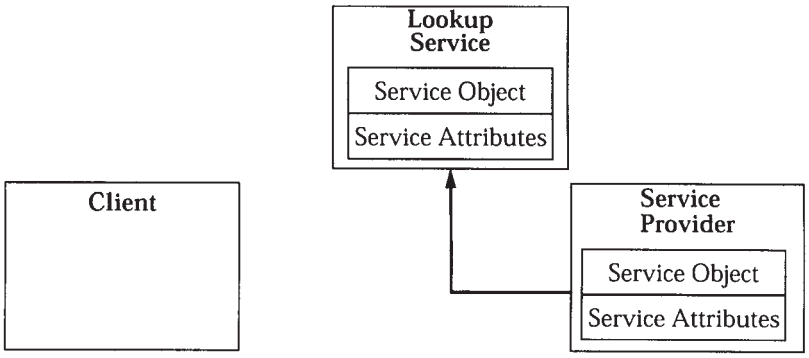


Abbildung 2: Join

ServiceInfo: enthält Namen, Hersteller, Verkäufer, Version, Modell und Seriennummer als String.

ServiceType: enthält ein Sinnbild (Icon), den anzuzeigenden Namen und eine Kurzbeschreibung.

Status: länger anhaltente Stati wie Fehler, Warnungen, Nachrichten

Jeder Service erhält eine eigene Nummer zur Erkennung vom Lookup-Service zugeteilt. Sie ist 128 Bit lang und durch Einarbeiten der Uhrzeit bei der ersten Anmeldung eineindeutig. Diese Kennung bleibt erhalten, bis sie unter Zuhilfenahme eines Administrators gelöscht wird. Der Service kann sich so zu einem späteren Zeitpunkt wieder anmelden und behält damit alle Einstellungen wie z.B. Zugriffsbeschränkungen.

Um Zugriffsbeschränkungen zu erteilen, muß ein Administrator nach der ersten Anmeldung am Lookup-Service die Zugriffsrechte angeben. Erst dann kann eine Ressource mit der gewünschten Sicherheit zur Verfügung gestellt werden.

Jeder Service kann sich sooft er es für nötig hält, anmelden. Ein Fernseher wird sich nur einmal anmelden - ein Verbindungsservice zwischen Intranet und Telefonleitung mit acht Anschlüssen wird acht Services anmelden. Somit wird die Verwaltung der Ressource dem Lookup-Service überlassen.

Durch den freigegebenen Code steht es den Unternehmen frei, den Lookup-Service so zu modifizieren, daß zeitlich fakturierte Nutzungskosten direkt an das Buchungssystem übergeben werden können. Auch ist es möglich Softwarekomponenten für eine bestimmte Zeit zu verleihen und dann entsprechend der Anzahl der Nutzungen abzurechnen. Das bedeutet eine vereinfachte Automatisierung der Abrechnungsverfahren und eine damit verbundene bessere Verrechnung von pagatorischen[1] Kosten.

2.3 Auffinden von Services

Ein potentieller Servicenehmer sucht ebenfalls mittels des erwähnten Broadcastmechanismus nach mindestens einem Lookup-Service. Diesen befragt er dann ob ein zugangsberechtigter Service, der verschiedene Anforderungen erfüllen kann, existiert. Der Lookup-Service wird

[1] Kosten, die sich aus der internen Leistungsverrechnung ergeben, denen aber keine Zahlung zugrunde liegt. Sie liegen oft über den wahren Kosten und sind nicht mit Marktpreisen für die Leistung vergleichbar. Beispiel: Der Einbau einer Druckerpatrone kostet pauschal 105,– DM unabhängig von der Einbaudauer und den Kosten der Patrone.

dem potentiellen Servicenehmer eine nicht unbedingt vollständige Liste verfügbarer Services übergeben.

Aus der Liste sucht der Servicenehmer meistens mit Hilfe eines Benutzers einen aus und läßt sich die Ressource reservieren. Dies geschieht mit Hilfe eines Lease, den der Servicenehmer vom Lookup-Service erhält. Dieser Lease enthält neben den administrativen Informationen wie Adresse, Eigenschaften und Dauer der Nutzungsberechtigung auch im Serviceobjekt Methoden, die den Service ansprechen. Diese Methoden können aus Einstellungsmasken und Übertragungsmechanismen bestehen.

Um einen Service aufzufinden muß der Servicenehmer dem Lookup-Service sagen, was er eigentlich will. Das geschieht durch eine Anfrage an alle ansprechbaren Lookup-Services. Dabei können verschiedene Parameter eingeschränkt werden. Jede im Serviceattribute enthaltene Information kann eingeschränkt werden.

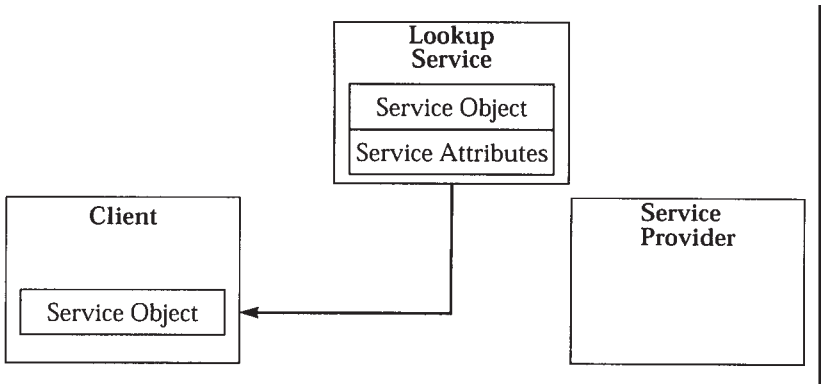


Abbildung 3: Lookup und Lease

Aufgrund der möglichen Größe der Antwort nimmt jeder Lookup-Service eine Reduktion der Antwortmenge vor, damit jeder mehrfach eingetragene Service nur einmal in der Liste erscheint.

Die Vereinigung aller Antwortmengen kann dann vom Servicenehmer dem Benutzer zur Auswahl vorgelegt werden. Ist ein Service zweifelsfrei ausgewählt, so wird beim Lookup-Service der Lease beantragt. Ein Lease enthält neben dem Zeitfenster, für das er gültig ist, das vom Servicegeber an den Lookup-Service übergebene Serviceobjekt.

2.4 Leasing

Kollisionen bei der Ressourcennutzung werden von Jini durch den Discovery and Join Mechanismus unterbunden. Jeder Service kommuniziert nur mit einem Servicenehmer, der auch einen Lease vorweisen kann. Den Lease erhält er vom Lookup-Service für eine vom Lookup-Service festgelegte Zeit ausgeliehen. In dieser Zeitspanne steht dem Servicenehmer die Ressource exklusiv zur Verfügung. Nach Ablauf der Zeitspanne ist keine weitere Kommunikation nötig. Alle Teilnehmer wissen, daß die Kommunikation jetzt beendet ist und der Lease erneut vergeben werden kann. Ein Lease kann vom Lookup-Service erneuert werden, die Erneuerung kann aber auch fehlschlagen.

Ein Lease enthält neben den statischen Informationen über den Service und der Zeitbeschränkung auch Methoden, die die Kommunikation zwischen Servicenehmer und dem Service übernehmen.

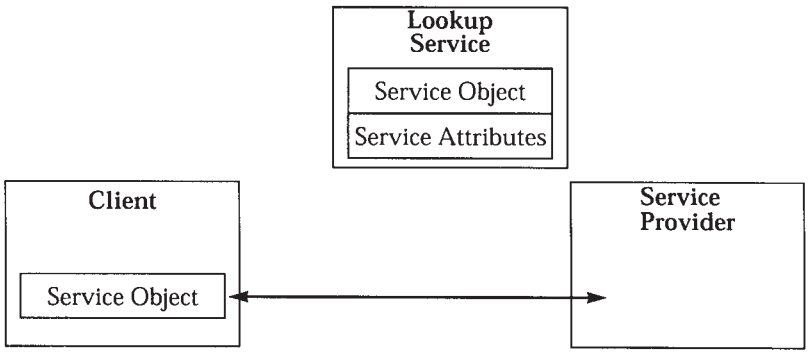


Abbildung 4: Kommunikation

2.5 Verbindungsaufbau – code injection

Damit der Servicenehmer nun den Service nutzen kann, muß er nun mit Hilfe des Lease mit dem Servicegeber Kontakt aufnehmen. Sun nennt die Übertragung des im Lease enthaltenen Serviceabjekts code injection. Dadurch wird ein Gerätetreiber just in time installiert und nach Benutzung wieder vernichtet. Die Softwaredistribution findet also zum letztmöglichen Zeitpunkt und zentral statt. Der Administrationsaufwand wird dadurch extrem reduziert. Zusätzlich steht durch die Implementierung in JAVA jedem Jini-Gerät der gleiche und mithin funktionierende Treiber zur Verfügung.

Der eigentliche Verbindungsaufbau kann jetzt mit den Methoden des Service Objekts erfolgen. Sie können jetzt die direkte Kommunikation und den Austausch der Daten regeln. Objekte können so noch beim Servicenehmer vorverarbeitet werden. Auf dem gleichen Weg können über das grafische User Intferface (GUI) Einstellungen vor oder während des Auftrags vorgenommen werden.

Diese vorteilhaften Eigenschaften eines Lease erwarten aber auch vom Servicenehmer, daß er Zugang zu einer Java Virtual Machine hat. Dadurch werden vor allem Geräte wie Haushaltsgeräte stark eingeschränkt: Sie würden sich unnötig verteuern. Deshalb hat Sun einige Vorschläge ausgearbeitet, die später dargestellt werden.

3 Aufbau von Jini-Teilnehmern

Wie bereits angesprochen besteht eine große Herausforderung der Jini-Technologie an die Endgeräte darin, daß jedes Gerät zumindest Zugang zu einer Java Virtual Machine haben muß. Nur dadurch ist gewährleistet, daß mittels code injection auch fremde Programme ausführbar sind.

Die Entwickler von Sun haben schnell erkannt, daß ein kompletter Rechner mit einer Java Virtual Machine eine Kaffemaschine zu teuer machen würde. Daher suchte man nach Empfehlungen um den Aufwand für die Virtual Machine zu reduzieren. Sun greift dabei jedoch auf bekannte Ansätze zurück.

3.1 Eingebaute Java Virtual Machine

Die ertse Möglichkeit besteht in je einer komplett in den Servicenehmer und den Servicegeber eingebauten Java Virtual Machine.

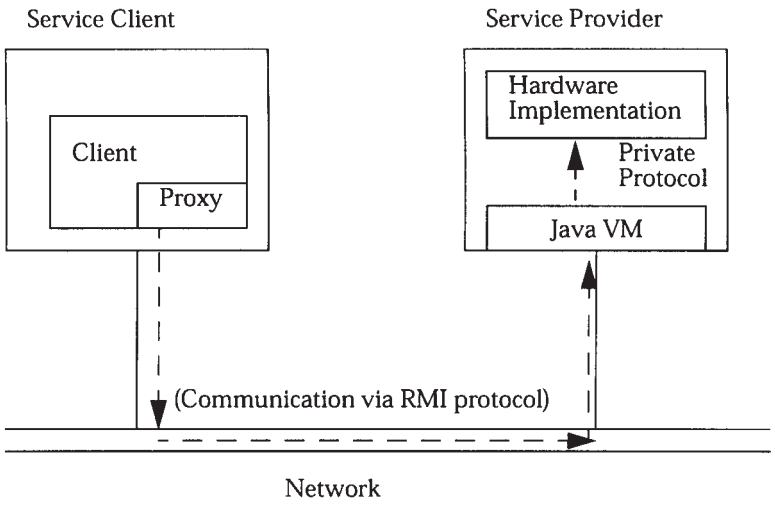


Abbildung 5: Eingebaute Java Virtual Machine

Dieser Ansatz eignet sich für alle Geräte, die sowieso schon einen Rechner besitzen, der die Kapazität hat eine Java Virtual Machine auszuführen. Dazu gehören Notebooks und PDAs[2].

Dieser Ansatz ist aber wegen der hohen Kosten ungeeignet für Geräte, die nur Statusinformationen an Servicetechniker weitergeben sollen oder für Geräte, die sehr einfache Funktionen ausüben sollen.

3.1.1 Eingeschränkte eingebaute Java Virtual Machine

Für Servicegeber, also Geräte, die einen Service zur Verfügung stellen, werden keine so hohen Anforderunegn an die Java Virtual Machine gestellt. Sie müssen lediglich einen bestimmten Satz von bereits bekannten Objekten ausführen können. Dazu gehören die Schnittstellen zum Lookup-Service aber auch Sicherheitsmechanismen und Speicher für mögliche Informationen.

Sun läßt jedoch keine Bestrebungen erkennen, eine solche Specialized Virtual Machine zu erstellen. Die Entwickler werden mit dieser Aufgabe alleine gelassen. In [Micr99f] wird deshalb auch auf klare Aussagen verzichtet.

3.2 gemeinsam genutzte Java Virtual Machine

Die Lösung der gemeinsam genutzten Java Virtual Machine stellt die Virtual Machine nur auf einem Gerät zur Verfügung. Andere Geräte können aber auf diese Maschine zugreifen. Sun erhofft sich durch die Reduktion der im Netz verfügbaren Java Virtual Machines einen Kostenvorteil.

Diese Idee verlangt aber nach einer Virtual Machine, die mit Anfragen von andern Geräten zurecht kommt. Zusätzlich müssen die Geräte über dasselbe Netzwerk kommunizieren, über das auch die eigentliche Kommunikation stattfindet.

[2] Auch WindowsCE rechner könnten eine JVM ausführen, wenn Microsoft wollte.

3.2.1 Hardwarelösung

Für manche Hausgeräte ist ein sogenannter Device-Bay empfohlen. Dabei handelt es sich um ein Gerät, daß die Kommunikation zwischen dem Netzwerk und den einzelnen Endgeräten eines Herstellers regelt (Gateway).

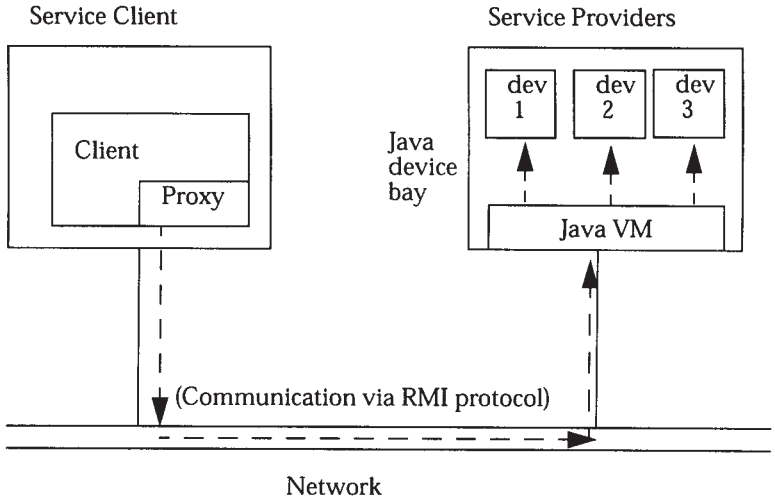


Abbildung 6: Eingebaute Java Virtual Machine

Die Endgeräte können über ein wahlfreies Netzwerk mit einem wahlfreien Protokoll mit der Device-Bay kommunizieren. Die Device-Bay setzt dann die Anfragen und Java-Aufrufe um und routet die Ergebnisse entweder durch das eigene Netz an einen Teilnehmer oder in das unternehmensweite Netzwerk.

3.2.2 Softwarelösung (Proxy)

Die Softwarelösung unterscheidet sich darin gegenüber der Hardwarelösung, daß die Geräte über das Netzwerk kommunizieren können, selbst aber keine Java Virtual Machine besitzen.

Sie kommunizieren über das Netzwerk mit einer Java Virtual Machine, die dann die Aufrufe für die Geräte ausführt. Auch hierbei ist mit einem erhöten Netzwerkaufkommen zu rechnen.

Bei diesem Ansatz ist es unerheblich, ob die Services und die Java Virtual Machine wirklich physikalisch getrennt sind.

Es ist möglich, daß der Servicenehmer zunächst über das Netzwerk seine Daten an die Java Virtual Machine schickt. Dort werden sie interpretiert und dann an eine zweite, dem Servicegeber zugeordnete JVM geschickt. Dort findet ggf. eine zweite Interpretation statt und die Daten werden abschließend an den Servicegeber selbst geschickt. Das Datenvolumen im Netzwerk wird also bei ungünstiger Implementierung eventuell vervielfacht. Für Videorecorder und Fernseher ist dies wegen der großen Datenmengen nicht zu empfehlen.

3.3 Java Virtual Machine via Internet Inter-Operability Protocol

Um schon entwickelte Funktionen und Services einbinden zu können, die nicht auf Java-Technologie beasieren, unterstützt Jini den Ansatz der Object Management Group (OMG) und deren Inter-Operability Protocol (IIOP). Eine Untermenge des RMI nutzt diese Fähigkeiten aus.

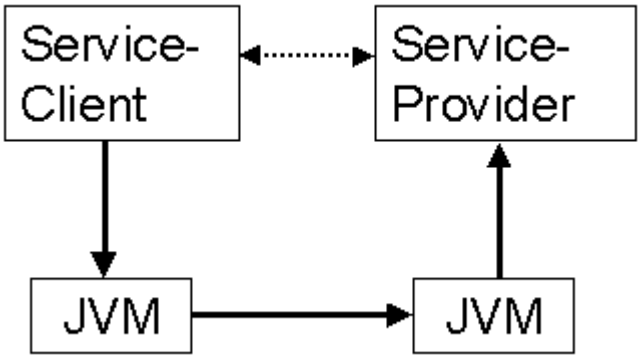


Abbildung 7: Verbindungsaufbau mit Proxy im gleichen Netzwerk

Geräte, die diesen Zugang nutzen wollen müssen in der Lage sein, einen IIOP direkt lesen und verarbeiten zu können. Das Gerät muß dann auch CORBA-Implementierung, ORB enthalten. Kennt das Gerät die möglichen IIOP-Pakete, die eintreffen können, kann es sie auch einfach interpretieren.

3.4 Umsetzung in Geräten

Da die Jini-Technologie sehr neu ist, wurde sie noch nicht in serienreifen Hardwareprodukten umgesetzt. Allerdings gibt es eine große Anzahl von Ankündigungen, wer alles Jini-Geräte zur Verfügung stellen will.

Im Softwaresektor ist eine Umsetzung dann sehr einfach, wenn ein schon bestehendes JAVA-Produkt integriert wird. Hier muß nur der kleine Teil der Jini-Objekte ergänzt werden.

3.4.1 Softwareumsetzung: Datek BusinessTone

Ein malaysianisches Softwarehaus, Datek[3], hat als erster Anbieter eine Jini-Anwendung präsentiert[4]. BusinessTone ist ein Enterprise Resource Management System.

Datek verteilt und verwaltet mit dem Produkt Softwarekomponenten. Der Benutzer zahlt nicht mehr für den Besitz von BusinessTone, sondern vielmehr für jede einzelne Nutzung von Teilkomponenten des Systems.

Jini kommt hier bei der Verteilung und Abrechnung der einzelnen Komponenten ins Spiel. Der Benutzer ruft eine Komponente auf. Nachdem das Serviceobjekt vom Lookup-Service innerhalb des Lease erhalten wurde, nimmt das Serviceobjekt zunächst Kontakt mit dem Abbuchungssystem von Datek auf. Dort wird die Nutzung angezeigt. Anschliessend werden die Komponenten ausgeführt.

Bis jetzt hat Datek das Produkt erst in einer Testumgebung eingesetzt. Ob BusinessTone erfolgreich ist wird sich noch zeigen. Vorteilhaft ist aber vor allem der durch Java und Jini verminderte Aufwand zur Versionspflege und die geringeren Verteilungskosten.

Für den Endanwender ist der Einsatz der Jini-Technologie im Softwareumfeld transparent.

[3] http://www.batavia.com/

[4] Das Softwarehaus ist nicht mit dem großen US-amerikanischen Billigbroker zu verwechseln.

3.4.2 Hardwareumsetzung: Quantum Massenspeicher

Wie schon erwähnt, ist die Umsetzung auf der Hardwareseite noch nicht so weit fortgeschritten, wie auf der Softwareseite. Trotzdem haben schon viele Hersteller Absichtserklärungen abgegeben. Zu diesen Unternehmen gehört auch der Massenspeicherhersteller Quantum.

Am Ansatz von JavaSpaces angreifend, können Programme einheitlich auf Massenspeicher wie zB. Festplatten zugreifen und die Informationen objektweise ablegen. Dabei wird auf serialisierte Objekte zurückgegriffen.

Die Massenspeicher sollen um die Funktionalität des Plug and Play im Netzwerkumfeld erweitert werden. Dazu wird Jini verwendet.

Jeder im Netz verfügbare Massenspeicher wird im Lookup-Service als Service verwaltet und steht so anderen Programmen zur Verfügung. Die Funktionalität von Fileservern teilen sich jetzt also der Lookup-Service und der Massenspeicher.

Ein Programm, das Zugriff auf einen mit JavaSpaces funktionierenden Massenspeicher benötigt, wird den entsprechenden Speicher also erst dann kennenlernen, wenn er das erste Mal aufgerufen wird.

4 Jini im Vergleich

Oft treten im Zusammenhang mit Jini Abkürzungen und Standards auf. Sun ist sehr darauf bedacht zwischen den einzelnen Ansätzen zu differenzieren und herauszustellen, daß es ein Miteinander mit allen Ansätzen gibt. Jini ist also nach Suns Auffassung eher ein Integrator verschiedener Ansätze von Sun und anderen.

SLP: SLP ist ein Protokoll zum dynamischen Auffinden von Services. Jini hat hier den Vorteil des plattformunabhängigen Codes. Eingeführt wurde das Protokoll von der SVRLock Working Group.

HAVi: ist eine Initiative für Heimnetzwerke von Sony und Philips. Jini arbeitet im HaVi Netzwerk und unterstützt diese Technologie.

JetSend: Im Gegensatz zu Jini ist JetSend von Hewlett Packard ein Austauschprotokoll für festgelegte Funktionen.

5 Sun Community Source License

Zusammen mit Jini hat Sun die Sun Community Source License veröffentlicht. Der Quellcode und große Teile der Dokumentation unterliegen diesem Vertrag. Die Lizenz ist ein Open-Source Ansatz - aber eben nicht komplett. Eine Auseinandersetzung mit den Rechten und Pflichten, die aus dem Vertrag entstehen, ist angebracht.

Sun teilt die Lizenznehmer in drei Gruppen auf:

Research Use License: Vertragsbedingungen, die sich auf das Erlernen, modifizieren und testen der Jini-Technologie beziehen.

Internal Deployment Use: Nutzung der Jini-Technologie im internen unternehmensweiten Umfeld.

Commercial Use License: Abkommen beim Verbreiten von Software außerhalb der Unternehmensgrenzen.

Alle Lizenznehmer sind Mitglieder der Sun Community. Diese Gemeinschaft ist Besitzer des Quellcodes und der Rechte an der Jini-Technologie.

5.1 Research Use License

Die Research Use Licence enthält den niedrigsten Grad der vertraglichen Bindung. Auf ihr bauen aber alle anderen Stufen auf. Die Lizenz besteht aus einer Sammlung von Rechten und Pflichten, denen der Lizenznehmer zustimmen muß, bevor große Teile der Dokumentation und der Quellcode sichtbar werden.

Die Sun Community sichert dem Lizenznehmer folgende Rechte zu:

- Nutzung und Änderung des Quellcodes
- Verteilung von Quellcode an Studenten und andere Lizenznehmer
- Verteilen von ausführbaren Dateien zu testzwecken
- Vorführen der Funktionalität des Quellcodes

Im Gegenzug verpflichtet sich der Entwickler, alle Fehlerkorrekturen, Modifikationen und Spezifikationen zu veröffentlichen und die Rechte daran der Sun Community zu übereignen. Der Lizenznehmer verpflichtet sich auch seinen Quellcode zu dokumentieren.

Darüber hinaus muß der Lizenznehmer Sorge dafür tragen, daß er nur Lizenznehmern Zugang zur Jini-Technologie gewährt. Informationen auf Webseiten müssen paßwortgeschützt sein und einen Hinweis auf die Lizenz enthalten.

5.2 Internal Deployment Use License

Die Internal Deployment Use License enthält zu den Regelungen der Research Use License noch das Recht, ausführbare Programme aus dem Quellcode zu erstellen und sie intern zu vertreiben.

Darüber hinaus verpflichtet sich der Lizenznehmer

- bestimmte Klassennamen nicht zu verwenden (java.*, jini.*, ...)
- die Unique Package Naming Convention einzuhalten
- Der Code muß einen Kompatibilitätstest von Sun absolvieren
- Erweiterungen müssen innerhalb von 90 Tagen veröffentlicht werden und frei von Urheberrechten sein.

5.3 Commercial Use License

Kommerzielle Nutzer sind darüber hinaus verpflichtet:

- auf die Verwendung des Jini-Codes hinzuweisen
- Innerhalb von 120 Tagen Upgrades des Jini-Quellcodes nachzuvollziehen
- Dem Kompatibilitätstest von Sun in seiner jeweils neuesten Version zu entsprechen

6 Zusammenfassung

Sun hat mit Jini ein Konzept vorgestellt, das die Bereitstellung von Diensten und Geräten in einem Netzwerk beinahe administrationsfrei regelt. Sekundärliteratur beurteilt diesen Ansatz jedoch mit unterschiedlichen Einschätzungen.

Ein Teil der Beurteilungen lobt den innovativen Ansatz und hält ihn uneingeschränkt für den besten, weil standardisierten, dokumentierten und frei zugänglichen Weg. Andere Stimmen verweisen hingegen darauf, daß die Gesamtheit des Systems in der heute benötigten Komplexität schon mit anderen Produkten abbildbar ist.

Alle Stimmen sehen aber die Zukunft von Jini in einer starken Abhängigkeit von der Unterstützung der Industrie und dem Vorhandensein von ersten Beispielanwendungen. Erst dann wollen viele das System abschließend beurteilen.

Jini zeigt aber gerade durch die Integration von bisherigen Teillösungen einen Weg auf, der Benutzern gefällt, weil der Wettbewerb der Hard- und Softwarehersteller auf deren Kernkompetenz gelenkt wird: Die Funktionalität der einzelnen Produkte rückt mehr in den Vordergrund, nicht die Art und Weise der Kontaktaufnahme mit dem Netz.

Zusätzlich müssen sich die Programmierer weniger Gedanken um gelöste Probleme machen - ein in Zukunft immer wichtiger Aspekt.

Literatur

[Micr99a] Sun Microsystems. *JavaSpaces Specification*, 1.0. Auflage, Januar 1999.

[Micr99b] Sun Microsystems. *Jini Architectural Overview - Technical Whitepaper*, Januar 1999.

[Micr99c] Sun Microsystems. *Jini Architectural Specification*, 1.0. Auflage, Januar 1999.

[Micr99d] Sun Microsystems. *Jini Community Licensing FAQ.* http://www.sun.com/jini/licensing/sclfaq.html, April 1999.

[Micr99e] Sun Microsystems. *Jini Connection Technology Demos.* http://www.sun.com/jini/demos/, April 1999.

[Micr99f] Sun Microsystems. *Jini Device Architecture Specification*, 1.0. Auflage, Januar 1999.

[Micr99g] Sun Microsystems. *Jini Discovery and Join Specification*, 1.0. Auflage, Januar 1999.

[Micr99h] Sun Microsystems. *Jini Discovery Utilities Specification*, 1.0. Auflage, Januar 1999.

[Micr99i] Sun Microsystems. *Jini Distributed Event Specification*, 1.0. Auflage, Januar 1999.

[Micr99j] Sun Microsystems. *Jini Distributed Leasing Specification*, 1.0. Auflage, Januar 1999.

[Micr99k] Sun Microsystems. *Jini Entry Specification*, 1.0. Auflage, Januar 1999.

[Micr99l] Sun Microsystems. *Jini Entry Utilities Specification*, 1.0. Auflage, Januar 1999.

[Micr99m] Sun Microsystems. *Jini Lookup Attribute Schema Specification*, 1.0. Auflage, Januar 1999.

[Micr99n] Sun Microsystems. *Jini Lookup Service Specification*, 1.0. Auflage, Januar 1999.

[Micr99o] Sun Microsystems. *Jini Technology Executive Overview*, 1.0. Auflage, Januar 1999.

[Micr99p] Sun Microsystems. *Jini Technology Frequently Asked Questions*, Januar 1999.

[Micr99q] Sun Microsystems. *Jini Technology Glossary*, 1.0. Auflage, Januar 1999.

[Micr99r] Sun Microsystems. *Jini Transaction Specification*, 1.0. Auflage, Januar 1999.

[Micr99s] Sun Microsystems. *Sun Community Source License*, Jini Technology Core Platform 1.0. Auflage, Januar 1999.

[uMar99] Andreas Zeidler und Marco Gruteser. Bezaubernde Geräte. *iX* (4), April 1999, S. 144–153.

MOBIWARE - Eine Plattform für verteilte Anwendungen mobiler Teilnehmer

Oliver Storz

Kurzfassung

In dieser Ausarbeitung sollen die Konzepte und Arbeitsweisen von MOBIWARE veranschaulicht werden. MOBIWARE ist eine Middlewareplattform, die die Integration mobiler Teilnehmer in ATM-Netze unterstützen soll. MOBIWARE stellt dazu ein umfassendes Softwarepaket zur Verfügung, zu dem unter anderem eine Vielzahl von Objekten gehört, welche je nach Bedarf zielgerichtet zu bestimmten Punkten im Netz ausgesendet werden können. Es wird dabei sehr stark auf die speziellen Anforderungen der einzelnen Anwendungen eingegangen. MOBIWARE unterstützt dabei vollständig die jeweiligen Dienstgüteanforderungen der Anwendungen, soweit dies der Netzzustand zuläßt. Besonders wird dabei auf die Adaptivität an wechselnde Bedingungen im Netzwerk Wert gelegt. Zu erwähnen ist ebenso die besonders einfache Erweiterbarkeit des MOBIWARE-Pakets.

1 Mobilnetze und MOBIWARE

Mobilität wird in unserer Gesellschaft immer wichtiger. Mobiltelefone und Notebooks unterstützen diese Entwicklung. Es fehlt aber bislang ein Konzept, mobile Rechner effizient in Netzwerke einzubinden. Es sind zwar durchaus die physikalischen Möglichkeiten, wie z.B. GSM-Adapter vorhanden, dabei wird jedoch häufig keine Rücksicht auf die speziellen Problematiken genommen, die im Zusammenhang mit Datenübertragung über Funkverbindungen entstehen. Hemmnisse sind dabei zu geringe Bandbreite der vorhandenen Mobilfunknetze bzw. zu schlechte Ausnutzung und Verteilung der zur Verfügung stehenden Kapazitäten, sowie eine relativ hohe Fehlerhäufigkeit. Soll beispielsweise von einem zentralen Server ein Video-Stream an verschiedene Rechner im Netz gesendet werden, wovon einige mobil, andere fest angebunden sind, werden die mobilen Teilnehmer aufgrund von fehlender Bandbreite auf der Funkstrecke wohl kaum in der Lage sein, den vollständigen Datenstrom ohne Abstriche zu empfangen. Man müßte also den Videostrom vom Server aus derart reduzieren, daß er von der Mobilstrecke in Echtzeit übertragen werden kann. Die daraus resultierenden Qualitätsabstriche würden dann aber alle Teilnehmer betreffen, also auch diejenigen, die fest ins Netz integriert sind und eigentlich in der Lage wären, das Video in vollem Umfang zu empfangen. Die in diesem Artikel vorgestellte Middlewareplattform MOBIWARE stellt nun eine Softwarelösung dar, die speziell auf die Bedürfnisse von Netzwerken mit mobilen Komponenten zugeschnitten wurde. MOBIWARE besteht hauptsächlich aus einer Vielzahl von Objekten, die je nach aktuellem Bedarf an den unterschiedlichsten Stellen in den einzelnen Komponenten des Netzes tätig werden können, um die aktuellen Anforderungen der mobilen Teilnehmer zu unterstützen, ohne dabei jedoch die fest angebundenen Teilnehmer einzuschränken.

Dabei besteht das typische Netz (Abbildung 1) aus folgenden Komponenten:

- Mobile Geräte

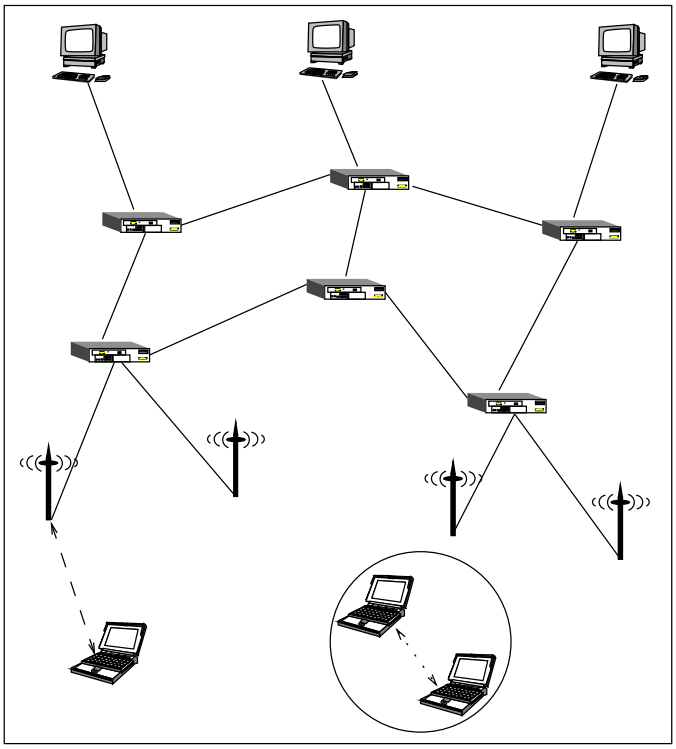


Abbildung 1: Ein beispielhaftes Netz für MOBIWARE.

- Netzzugangspunkte für mobile Geräte in Form von Funkstationen
- ein fest verkabeltes ATM-Netz mit Switches/Routern, Servern und Workstations

Dabei können mobile Geräte entweder mittels Netzzugangspunkten in das Gesamtnetz eingebunden werden, oder durch Zusammenschluß mehrerer mobiler Einheiten ein eigenständiges Netz bilden. MOBIWARE kann in allen oben genannten Stellen im Netzwerk zum Einsatz kommen, je nachdem, wie viel Funktionalität gerade zur Erfüllung der aktuellen Aufgabe erforderlich ist. MOBIWARE wurde mit folgenden Hauptzielen entwickelt:

1. Reduzierung der Ausfallwahrscheinlichkeiten beim Wechsel von einer Mobilfunkzelle in eine andere
2. Verbesserung der Ressourcennutzung in drahtlosen Netzen
3. Angemessene Anpassung von Datenströmen in mobilen Netzen mit ständig wechselnder Übertragungsqualität

Ziel dieser Ausarbeitung ist es, diese Konzepte näher zu erläutern, sowie diese anhand von einigen Beispielen zu verdeutlichen.

2 Adaptivität

MOBIWARE stellt mobilen Anwendungen auf der Transportschicht Funktionen zur Dienstgüteadaption zur Verfügung. Im folgenden werden die zwei wichtigsten beschrieben:

- Das *Setzen einer Nutzenfunktion*. Die Nutzenfunktion beschreibt die Anpassungsfähigkeit einer Anwendung mit zugehörigem Datenstrom an Veränderungen der Bandbreite. Sie gibt an, wie viel Nutzen, wenn überhaupt, die Anwendung aus einer gewissen Vergrößerung der Bandbreite ziehen kann, bzw. wie viel Verschlechterung der Qualität eine Verringerung der Bandbreite bedeutet. Aus der Nutzenfunktion kann man den minimalen Bandbreitenbedarf einer Applikation bestimmen. Bezüglich der Anpassungsfähigkeit an Bandbreitenschwankungen kann man folgende Anwendungsklassen unterscheiden:

  1. stark adaptive Anwendungen: Sie reagieren stetig auf Bandbreitenveränderungen und nähern sich asymptotisch an einen maximalen Nutzenwert an.
  2. schwach adaptive Anwendungen: Diese Anwendungen reagieren ebenfalls stetig auf Bandbreitenveränderungen, können jedoch bis zu einer gewissen Bandbreite nur wenig Nutzen aus den zusätzlich zugewiesenen Ressourcen ziehen, danach wächst allerdings bei weiterer Bandbreitenerhöhung der Nutzen für die Anwendung exponentiell an.
  3. linear adaptive Anwendungen: Anwendungen, die eine lineare Nutzenfunktion besitzen.
  4. diskret adaptive Anwendungen: Die Nutzenfunktion dieser Anwendungen verläuft treppenstufenförmig. Bei dieser Art von Datenströmen macht es Sinn, freiwerdende Ressourcen im Netzwerk der Anwendung nur zur Verfügung zu stellen, wenn diese aufgrund ihrer diskreten Anpassungsfähigkeit auch wirklich davon profitieren kann. Ein Beispiel hierfür sind die Video-Streams, die in der Testumgebung von MOBIWARE zum Einsatz kommen. Dabei wird das Video im Server in mehrere unterschiedliche Qualitätsstufen zerlegt, die sogenannten Layers. Die geringste Qualität bzw. Auflösung entspricht dabei der sogenannten Baselayer. Diese entspricht der minimalen Bandbreite, die nötig ist, um das Video zu übertragen. Zusätzlich werden sogenannte Enhancementlayers zur Verfügung gestellt, die bei verfügbarer Bandbreite zu einer höheren Qualität des Videos beim Empfänger führen.

- Das *Angeben einer Adaptionspolitik*. Sie beschreibt, wann und damit auch wie häufig auf Veränderungen der verfügbaren Bandbreite reagiert werden soll. So kann es zum Beispiel den Wünschen der Anwendung entsprechen, häufige Schwankungen um kleine Beträge zu vermeiden. Man würde dann freiwerdende oder knapperwerdende Bandbreite einer solchen Anwendung nur in Sonderfällen zuweisen bzw. entziehen. In einem anderen Fall wäre es vielleicht wünschenswert, alle verfügbaren Ressourcen sofort nutzen zu können. Man würde eine solche Anwendung dann schneller von Veränderungen profitieren lassen. Insgesamt kann man bezüglich der Adaptionspolitik vier verschiedene Einstellungen unterscheiden:

  1. schnelle Adaption: Es soll sofort auf Veränderungen reagiert werden.
  2. langsame Adaption: Man reagiert erst nach Ablauf einer gewissen Pufferzeit.
  3. beim Zellenwechsel: Auf Bandbreitenveränderungen wird nur beim Wechsel von einer Funkzelle in eine andere reagiert.
  4. niemals: Nachdem beim Verbindungsaufbau für die Anwendung eine gewisse Bandbreite reserviert und zugewiesen wurde, wird diese nicht mehr verändert.

3 MOBIWARE-Objekte

MOBIWARE basiert auf dem Einsatz verteilter Objekte, die hauptsächlich mittels Java und xbind [ALLM96] realisiert sind. Xbind ist eine auf CORBA aufbauende Plattform, die entwickelt wurde, um anwendungsspezifische Dienstgüteanforderungen in Netzwerken in allen

Schichten zu unterstützen. Auf der Transportschicht beinhaltet MOBIWARE Objekte, um die Übertragung von Audio-, Video- und sonstigen Echtzeitanwendungen im Netzwerk zu erleichtern. Diese Objekte sind in Java realisiert und werden beispielsweise in mobilen Rechnern sowie in Netzzugangspunkten eingesetzt. Zu ihren Aufgaben gehören zum Beispiel

- das Zerlegen sowie das Zusammenfügen von Datenströmen,
- die Überwachung von Datenströmen,
- die Kontrolle der Übertragungsraten sowie der Abspielraten von Multimediaanwendungen,
- die Überwachung von Ressourcen sowie
- das Puffermanagement

Außerdem gibt es noch zwei spezielle Objekte, die im Gegensatz zu den oben genannten aktiver Natur sind:

- Aktive Medienfilter: Diese werden typischerweise in den Netzzugangspunkten, also den Mobilfunksendestationen eingesetzt. Sie sorgen dafür, daß bei Multimediaübertragungen nur soviel vom gesamten Netzwerkstrom über die Funkstrecke übertragen wird, wie überhaupt Bandbreite für die Anwendung zur Verfügung steht. Dies wird dadurch erreicht, daß überflüssige Enhancementlayers einfach aus dem Strom gefiltert und nicht mehr weitergesendet werden.
- Adaptive Fehlerkorrekturfilter: Diese sollen Datenströme gegen Störungen bei der Übertragung per Funk schützen. Dabei soll der Anteil der Fehlerkorrekturinformationen im Datenstrom an die jeweiligen Störungsverhältnisse angepaßt werden, um bei guter Übertragung nicht unnötig Fehlerkorrekturmaßnahmen durchzuführen. Bei schlechten Verhältnissen soll aber auch die Möglichkeit bestehen, mehr Sicherungsmaßnahmen zu ergreifen.

Weitere Informationen zum Thema “aktive Filter” sind in der entsprechenden Literatur [BaAK97] zu finden.

Auf der Vermittlungsschicht kommen hauptsächlich xbind-Objekte zum Einsatz. Diese bilden ein programmierbares mobiles Netzwerk. Programmierbar heißt dabei, daß die Funktionalität der Schicht einfach durch Verwendung der vorhandenen Objekte und das Einfügen neuer Objekte erweitert werden kann. Die Vermittlungsschicht besteht hauptsächlich aus einer Menge von Objekten, deren Aufgabe es ist, Geräte im Netzwerk zu abstrahieren, sowie aus Proxy-Objekten, die verschiedene Aspekte der Adaptivität von MOBIWARE unterstützen sollen.

Das Besondere auf der Sicherungsschicht ist die programmierbare MAC-Schicht [BiCa98]. Diese stellt eine Reihe von Grundfunktionen für den vernünftigen Einsatz von mobilen Anwendungen mit variablen Dienstgüteanforderungen zur Verfügung. Dabei kann sie ebenso wie die Vermittlungsschicht einfach erweitert und so auf die Anforderungen neuer Anwendungen angepaßt werden.

Es folgt eine kleine Übersicht über die einzelnen Objekte und Klassen, welche sie sich in der MOBIWARE-Distribution befinden:

- *MobileDevice*: Repräsentiert ein mobiles Gerät. Es stellt Funktionen zum Abfragen von Statusinformationen, zur Registrierung mit einem neuen Netzzugangspunkt, zum Aufbau von Netzwerkverbindungen, sowie zum Beantragen einer neuen Dienstgüte zur Verfügung. Die Zustandsinformationen dieses Objekts beinhalten hauptsächlich die Dienstgütespezifikationen der von diesem Objekt empfangenen und gesendeten Datenströme. Außerdem beinhaltet das MobileDevice Funktionen, um z.B. den Fehlerkorrekturlevel oder die Skalierung von Videoströmen im Zugangspunkt dynamisch zu setzen und zu verändern.

- *AccessPoint*: Der Netzzugangspunkt. Hier stehen Methoden zur Anbindung von mobilen Stationen an Objekte im verkabelten Netzteil zur Verfügung. Außerdem spielt dieses Objekt beim Zellenwechsel sowie beim Einbringen von aktiven Transportobjekten, z.B. von aktiven Medienfiltern, eine entscheidende Rolle.

- *NodeServer*: Dieses Objekt stellt ATM-Switches und Router dar. Seine Hauptaufgaben sind das Bereitstellen von Namensbereichen und Netzwerkressourcen. Es beinhaltet außerdem Verbindungsinformationen für die über diesen Knoten laufenden Netzwerkverbindungen.

- *MobilityAgent*: Er ist zuständig für die Adaption von Datenströmen und übernimmt Managementfunktionen für mobile Einheiten. Für Adaption und Management wird mit NodeServer-Objekten zusammengearbeitet.

- *ATMRoute*: Dieses Objekt spielt ebenfalls eine entscheidende Rolle beim Zellenwechsel. Es stellt Routinginformationen bereit und interagiert mit dem MobilityAgent-Objekt.

- *MWManage*: Mit Hilfe dieses grafischen Managementtools können vielerlei Einstellungen im gesamten Netzwerk vorgenommen werden. So können zum Beispiel Datenströme eingerichtet werden, das Bündeln von Datenströmen (siehe Abschnitt 4) ein- bzw. ausgeschaltet werden, sowie erzwungene Zellenwechsel herbeigeführt werden. Es ist ebenso möglich, direkt in die Arbeit von aktiven Medienfiltern und adaptiven FEC-Filtern einzugreifen.

- *FilterDaemon*: FilterDaemons ermöglichen das Einbringen von aktiven Filtern in Form von Java-Bytecode in Zugangspunkte hinein. Diese Filter werden danach von dem FilterDaemon auch ablauffertig gemacht und konfiguriert.

- *FilterServer*: Hier werden in einer Art Datenbank die verfügbaren Filter bereitgehalten, von wo aus sie dann in jeden beliebigen strategischen Punkt im Netzwerk transportiert werden können.

- *MediaSelector*: Diese Klasse implementiert einen aktiven Filter für MPEG-1 Videoübertragungen. Es werden entweder der gesamte Videodatenstrom, oder Baselayer und Enhancementlayer 1, oder nur die Baselayer weitergeleitet. Instanzen der MediaSelector-Klasse können dynamisch in Netzzugangspunkte eingebracht werden, je nachdem, wo sie gerade benötigt werden.

- *MediaSource*: Eine Quelle für verschiedene Multimediaübertragungen über ATM-Netze.

- *MediaReceiver*: Ein Multimediaclient. Dieser wird, sofern er in einem mobilen Gerät zum Einsatz kommt, auf das dort schon vorhanden MobileDevice-Objekt aufgesetzt.

Im folgenden soll das Zusammenspiel verschiedener Objekte anhand des Einbringens eines aktiven Filters in einen Zugangspunkt beschrieben werden.

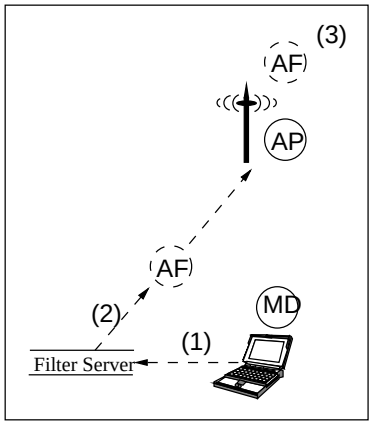


Abbildung 2: Zusammenspiel der MOBIWARE-Objekte beim Einbringen eines aktiven Filters.

Das MobileDevice-Objekt (MD) entscheidet, daß der Einsatz eines aktiven Filters (AF) im Zugangspunkt nötig wird. Es wählt daraufhin einen geeigneten Filter beim FilterServer aus (1), transportiert diesen zum AccessPoint (AP) (2), wo er geladen und initialisiert wird (3). Alle weiteren Anpassungen werden vom Filter selbst vorgenommen.

4 Zellenwechsel und Flow-Bundling

Eines der größten Probleme bei der Verwendung mobiler Rechner, die über Funkzellen an verdrahtete Netze angebunden sind, ist der Wechsel eines Gerätes von einer Zelle zu einer anderen.

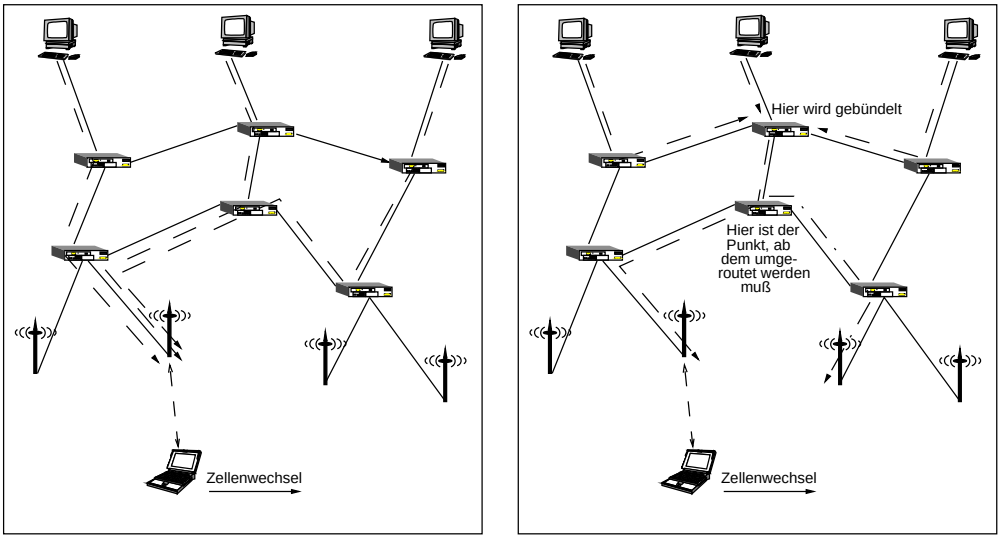


Abbildung 3: Zellenwechsel mit und ohne Flow-Bundling

Um diesen Wechsel zu vereinfachen, wurde das Konzept des Flow-Bundlings, also des Bündelns von Datenströmen, eingeführt. Dabei werden so viele Datenströme wie möglich, die zu einer mobilen Einheit hinführen, zu einem einzigen Strom zusammengefaßt. Das Zusammenfassen selbst sollte so früh wie möglich, d.h. so nahe wie möglich an den Ursprüngen der Datenübertragung geschehen. Der Vorteil liegt auf der Hand. Wechselt ein mobiler Rechner von einer Mobilfunkzelle zur nächsten, so muß nur für einen Datenstrom ein neuer Weg durch das Netzwerk gefunden werden. Die Zeitersparnis dabei ist immens. MOBIWARE erlaubt

allerdings auch das Deaktivieren des Flow-Bundlings. Dies ist zum Beispiel für Testzwecke interessant. Doch nun zum eigentlichen Zellenwechsel. Dabei gibt es zwei mögliche Ansätze:

- Den sogenannten "harten Wechsel": Dabei wird zuerst die Verbindung mit der bisherigen Basisstation unterbrochen und erst dann eine neue Verbindung mit der neuen Zelle aufgebaut.
- Den "weichen Wechsel": Hier werden für eine gewisse Übergangszeit sowohl eine Verbindung zur alten Funkzelle als auch eine zur neuen Zelle aufrechterhalten. Erst wenn die Verbindung zur neuen Zelle eine gewisse Mindestqualität erreicht hat, wird die Verbindung zur alten abgebrochen.

Bei MOBIWARE kommt eine Mischung aus hartem und weichem Wechsel zum Einsatz. Zuerst einmal ist zu bemerken, daß die Initiative zum Herstellen einer Verbindung mit dem neuen Zugangspunkt, sowie zum Abbruch der Verbindung mit dem alten Punkt, vollständig von der mobilen Einheit ausgeht. Dabei werden aus dem kabelgestützen Netzwerk kommende Datenströme eine gewisse Zeit lang sowohl vom alten als auch vom neuen Zugangspunkt empfangen, was dem Konzept des weichen Wechsels entspricht. Von der mobilen Einheit ausgehende Datenströme werden jedoch nach dem Prinzip des harten Wechsels von einer Funkzelle an die andere weitergereicht. Im Prinzip ist der Zellenwechsel jedoch vollständig frei konfigurierbar. Obiger Ablauf stellt also nur die momentan verwendete Version dar, es ist dem Anwender jedoch freigestellt, andere Kombinationen für seine spezifischen Anwendungsproblematiken festzulegen.

Um einen korrekten Wechselvorgang einzuleiten, muß die mobile Einheit natürlich zuerst einmal über benachbarte Sende- und Empfangsstationen in Kenntnis gesetzt werden. Dies geschieht über ungerichtete Beacon-Signale, einer Art Funkfeuer, die ständig von den Netzzugangspunkten abgestrahlt werden. Diese beinhalten Informationen über die momentan verfügbare Bandbreite in der Senderstation. Das Beacon-Signal dient außerdem als Testsignal für die momentane Übertragungsqualität auf der entsprechenden Funkstrecke. Fällt die Dienstgüte auf der Verbindung zum momentan verwendeten Zugangspunkt, wählt die Software im mobilen Gerät einen besseren Zugangspunkt aus und beginnt mit dem Zellenwechsel.

Dabei werden folgende Schritte nacheinander durchlaufen (Abbildung 4):

1. Das MobileDevice-Objekt (MD) empfängt ständig die Beacon-Signale der es umgebenden Funkzellen.
2. Wird die Verbindung zum alten Netzzugangspunkt zu schlecht, beginnt das mobile Gerät mit der Registrierung bei einem neuen Zugangsknoten, der zuvor aufgrund seiner Beacon-Informationen als geeignet ausgewählt wurde.
3. Das AccessPoint-Objekt (AP) im Zugangsknoten setzt sich daraufhin mit einem MobilityAgent-Objekt (MA) in Verbindung, das sich prinzipiell in jedem beliebigen Punkt im Netz befinden kann.
4. Das MobileDevice gibt dem neuen Zugangsknoten die Anweisung, mit den Wechseloperationen zu beginnen. Als Parameter werden dabei die Identifikatoren der aktuell empfangenen Datenstrombündel übergeben.
5. Das AccessPoint-Objekt setzt sich mit dem zuvor kontaktierten MobilityAgent in Verbindung und reicht an diesen die Identifikatoren der neu zu routenden Datenstrombündel weiter.

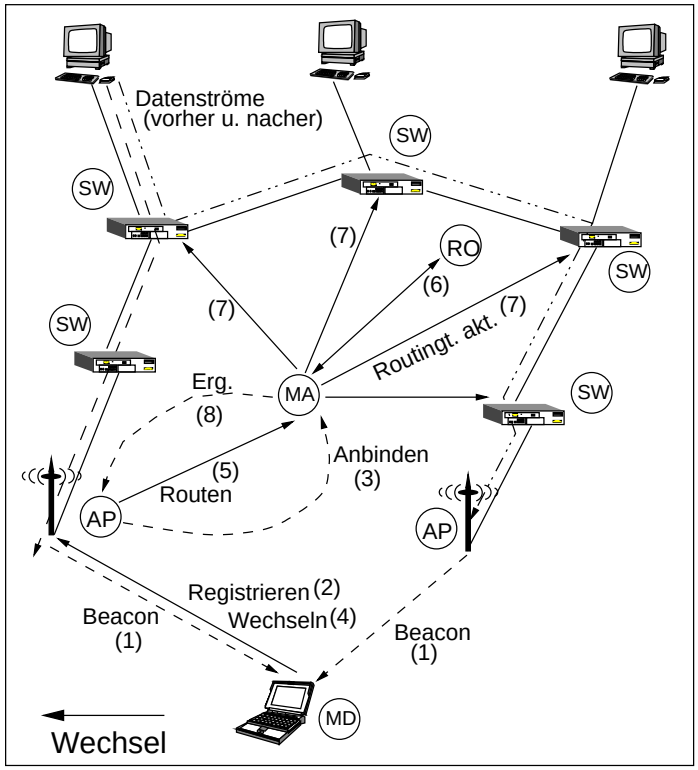


Abbildung 4: Kontrollfluß beim Wechsel von einer Mobilfunkzelle in eine andere.

6. Der MobilityAgent bezieht Routinginformationen von einem ATMRoute-Objekt (RO), um die Lage derjenigen Switches bzw. Router zu finden, denen neue Befehle erteilt werden müssen.

7. Danach werden die einzelnen NodeServer-Objekte (SW), also die abstrakten Switches/Router, benachrichtigt, worauf diese den entsprechenden neuen Pfad in die Routingtabellen eintragen. Da das hierzu verwendete Managementprotokoll kein Flow-Bundling unterstützt, müssen diese Veränderungen für jeden Datenstrom einzeln durchgeführt werden. Das Benachrichtigen der NodeServer durch den MobilityAgent ist also der letzte Schritt, in dem Flow-Bundling verwendet werden kann. Natürlich muß im Anschluß an die gesamte Prozedur auch noch für das Reservieren von Bandbreite für die neuen Datenverbindungen, sowie für das Konsistenthalten der Rechnernamen der mobilen Geräte im Netz gesorgt werden.

8. Das MobilityAgent-Objekt benachrichtigt anschließend den AccessPoint über die neu verfügbare Dienstgüte, sowie über weitere Parameter der Verbindung. Anschließend arbeitet es noch mit entsprechenden Netzobjekten zusammen, um das Einbringen von aktiven Medienfiltern zu erwirken.

Man beachte, daß die mobile Einheit die Verbindung zum alten Netzzugangspunkt nicht explizit abbricht. Vielmehr wird die Verbindung nach dem Ausbleiben regelmäßiger Refresh-Nachrichten (siehe Abschnitt 5) abgebrochen, wobei auch die entsprechenden MOBIWARE-Objekte z.B. aus dem Zugangsknoten entfernt werden.

4.1 Testergebnisse mit einem Datenstrom

Messungen der Dauer eines Zellenwechsels, die in der Testumgebung der Entwickler von MOBIWARE mit nur einem Datenstrom durchgeführt wurden, ergaben folgende Werte:

normaler Code	Nach Verbesserungen durch		
	das Zentralisieren von Objekten	Reduzierung der CORBA-Aufrufe	Pre-Binding
171 ms	111 ms	93 ms	51 ms

Es wird sehr deutlich, daß die ursprüngliche Dauer eines Wechsels von einem Zugangspunkt zum anderen von 171 ms auf 51 ms deutlich verbessert wurde. Besonders das Zentralisieren von verteilten Objekten, sowie das sogenannte Pre-Binding tragen erheblich zu einer Beschleunigung des gesamten Wechselprozesses bei.

Dabei wurde im ersten Fall durch das Vereinigen vorher im mobilen Gerät ansässiger Objekte, die jetzt im Netzzugangspunkt angesiedelt wurden, die Anzahl langsamer CORBA-Aufrufe über die Funkstrecke hinweg verringert.

Das Konzept des Pre-Bindings hingegen baut auf dem Konzept auf, die entsprechenden Verbindungsaufnahmen zwischen den am Zellenwechsel beteiligten Objekten schon lange vor der eigentlichen Wechseloperation zu tätigen. Erste Testergebnisse mit dem normalen MOBIWARE-Code ergaben nämlich, daß sich die gemessenen 171 ms aufteilen lassen in 30 ms, die für den Aufbau von Funkverbindungen verwendet wurden, 35 ms für den Aufbau von kabelgestützten Netzverbindungen, sowie ganze 102 ms für die Suche und Verbindungsaufnahme zwischen verteilten Objekten im Netz. Mit Pre-Binding wird dieser Prozeß vom mobilen Gerät initiiert, sobald es Beacon-Signale von benachbarten Funkstationen empfängt. Es werden dann Vorbereitungen für den Wechsel mit allen benachbarten Stationen getroffen, woraufhin diese selbst Verbindung mit den benötigten Objekten im Netz aufnehmen. Stellt sich nun eine Situation ein, bei der ein Wechsel vorgenommen werden muß, sind alle Vorbereitungen bereits getroffen, der Wechsel selbst kann also mit einem Minimum an Funktionsaufrufen schnell und reibungslos bewältigt werden. Allerdings ist besonders bei sehr großen Netzen im Vorfeld mit einem erheblichen Mehraufwand durch die Verwendung von Pre-Binding zu rechnen. Eventuell wäre es hier sinnvoll, der Anwendung zu überlassen, ob Pre-Binding verwendet werden soll, oder nicht. Wechselt ein mobiles Gerät z.B. nie in eine andere Zelle, und ist dieses Verhalten schon frühzeitig bekannt, dann macht es aufgrund des dazu nötigen Aufwands wenig Sinn, Pre-Binding zu betreiben. Ist allerdings abzusehen, daß im Laufe der gesamten Anwendungsdauer mehrere Zellenwechsel stattfinden werden, so sollte es möglich sein, diese mit Pre-Binding zu beschleunigen.

4.2 Testergebnisse mit mehreren Datenströmen

Hier wird die Bedeutung von Flow-Bundling deutlich. Folgenden Tabelle zeigt die gemessenen Zeiten, die ein Zellenwechsel in der Testumgebung der MOBIWARE-Entwickler benötigte, jeweils mit und ohne Flow-Bundling, sowie mit und ohne die im vorigen Abschnitt beschriebenen Verbesserungen. Die Verbesserungen beinhalten dabei auch Veränderungen am beim Routing verwendeten Managementprotokoll, um das Konzept des Flow-Bundling (FB) besser zu unterstützen.

Codevariante	Anz. Datenströme	Wechselzeiten in ms		Verbesserung in %
		ohne FB	mit FB	durch FB
normaler Code	2	250	200	20
verbesserter Code	2	67	56	16
normaler Code	10	780	320	59
verbesserter Code	10	420	155	63

Man sieht, daß die Verwendung von gebündelten Datenstömen immer zu einer Vereinfachung des Wechselvorgangs führt. Besonders deutlich wird der Unterschied bei der Verwendung

des verbesserten MOBIWARE-Codes. Eine Verbesserung von 63 % im Vergleich zur gleichen Codeversion ohne Flow-Bundling läßt bezüglich der Vorteile dieser Technologie keinen Zweifel mehr zu.

5 Soft-State

Auch ein Netzwerk, in dem MOBIWARE eingesetzt wird, ist ständigen Veränderungen unterworfen. Mobile Geräte nehmen erstmalig Verbindung mit Zugangspunkten auf und verlassen diese wieder. Datenströme müssen umgeroutet werden. Die in einer Funkzelle verfügbare Bandbreite wechselt ständig. Das gleiche gilt für die verfügbaren Bandbreiten im festen ATM-Netzwerk. Auch hier entstehen ständig neue Lastverhältnisse. Datenverbindungen müssen auf- und abgebaut werden. Ressourcen müssen reserviert und später wieder freigegeben werden. Diese ständigen Veränderungen verlangen nach einem Neztwerk, das sich dynamisch an die gerade herrschenden Verhältnisse anpassen kann.

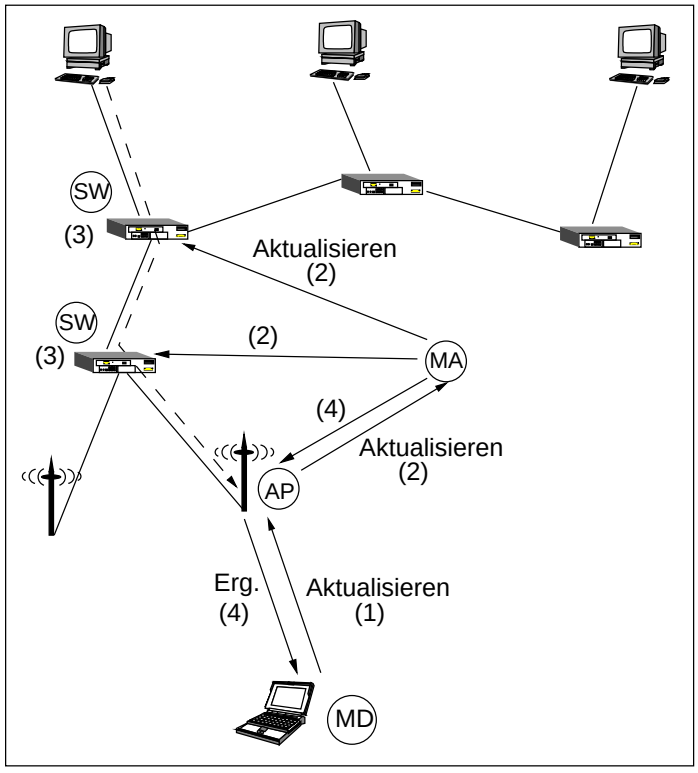


Abbildung 5: Aktualisierungsablauf im Beispielnetz.

Soft-State bietet einen Ansatz, diese Problematik in den Griff zu bekommen. Die mobile Einheit sorgt in regelmäßigen Abständen dafür, daß alle ein Datenbündel betreffenden Werte in allen dabei beteiligten Objekten aktualisiert werden. Dieser Mechanismus erlaubt es dem Netzwerk, die verfügbare Bandbreite kontinuierlich fair zu verteilen. Der genaue Ablauf sieht folgendermaßen aus:

1. Der mobile Rechner (MD) schickt dem aktuellen AccessPoint-Objekt (AP) den Befehl, ein bestimmtes Datenstrombündel zu aktualisieren. Die Nachricht enthält dabei u.a. den Namen des Bündels, dessen Bandbreitenanforderungen, seine Bandbreite-Nutzen-Funktion sowie die zu verwendende Adaptionspolitik.

2. Der AccessPoint sendet seinerseits einen Aktualisierungsbefehl an das zugehörige MobilityAgent-Objekt (MA), was seinerseits Meldungen an die beteiligten NodeServer (SW) schickt.

3. Die NodeServer setzen ihren Timer zurück, bei dessen Ablauf die Verbindung abgebaut worden wäre. Sie prüfen dann, wieviel Bandbreite sie der Datenverbindung auf dem betreffenden Netzwerkabschnitt in Abhängigkeit von den Bedürfnissen des aktuell betrachteten Datenstroms sowie anderen Datenströmen zur Verfügung stellen können. Anschließend setzen sie den MobilityAgent über verfügbare Bandbreite und sonstige relevante Parameter in Kenntnis. Dieser meldet dann die aktuell verhandelte Dienstgüte, nämlich das Minimum der von den NodeServern erhaltenen Dienstgütedaten, an den AccessPoint zurück.

4. Der AccessPoint setzt das mobile Gerät über die auf der Funkstrecke aktuell vorhandene Dienstgüte in Kenntnis.

Zur Adaption an schwankende Bandbreiten steht derzeit in MOBIWARE nur das 3-stufige Adaptionskonzept von Baselayer, Enhancementlayer 1 und Enhancementlayer 2 zur Verfügung.

5.1 Beispiel

In der Testumgebung von MOBIWARE wurden umfassende Tests durchgeführt, wobei als Testdaten hauptsächlich Videos übertragen wurden. Abbildung 6 zeigt den typischen Ablauf der kontinuierlichen Anpassungen entsprechend der Soft-State Idee.

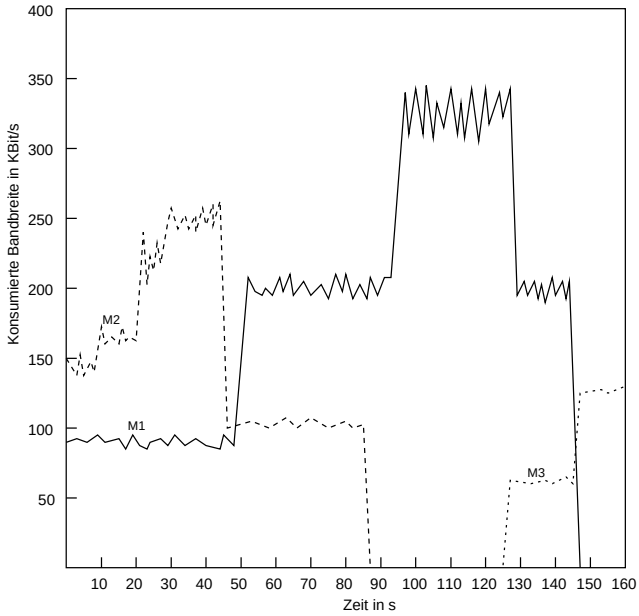


Abbildung 6: Testergebnisse des ersten Versuchs zu Soft-State

Zu Beginn des Versuchs befinden sich zwei mobile Geräte in ein und derselben Funkzelle. Gerät M1 empfängt das Video “True Lies”, Gerät M2 eine Folge der “Star Wars”-Trilogie. M1 empfängt dabei nur eine Baselayer des Videos mit 80 kBit/s, während M2 sein Video mit 150 kBit/s empfängt, was Baselayer plus einer Enhancementlayer entspricht. Insgesamt stehen 330 kBit/s zur Verfügung. Im Laufe der Zeit bewegt sich M2 immer weiter vom Sender weg, was zu einer erhöhten Tätigkeit der Fehlerkorrektur im Netzzugangspunkt und damit auch

zu einer immer weiter steigenden Bandbreitennutzung durch M2 führt. Nach insgesamt 50 s übersteigen die Anforderungen von M2 die verfügbare Bandbreite, worauf der Datenstrom, der zu M2 führt, auf die Baselayer zurückgestuft wird. M2 konsumiert nun die 80 kBit/s der Basisschicht sowie zusätzlich rund 20 kBit/s, die für Fehlerkorrekturmaßnahmen verwendet werden. Die freigewordenen Ressourcen sind nun dem Datenstrom von M1 zugewiesen worden. M1 empfängt nun Baselayer und erste Enhancementlayer des Videos. Nach 85 s verläßt M2 die gemeinsame Funkzelle, woraufhin M1 sogar in den Genuß der zweiten Enhancementlayer kommt und damit das komplette Video ohne Abstriche empfängt. Diese Phase geht allerdings zu Ende als sich bei 125 s ein weiteres Gerät (M3, empfängt HTTP-Daten) in der Funkzelle anmeldet, worauf M1 nur noch Baselayer und Enhancementlayer 1 empfängt. M3 kann schließlich nach dem Zellenwechsel von M1 mit der nun verfügbaren Bandbreite eine zweite HTTP-Verbindung öffnen.

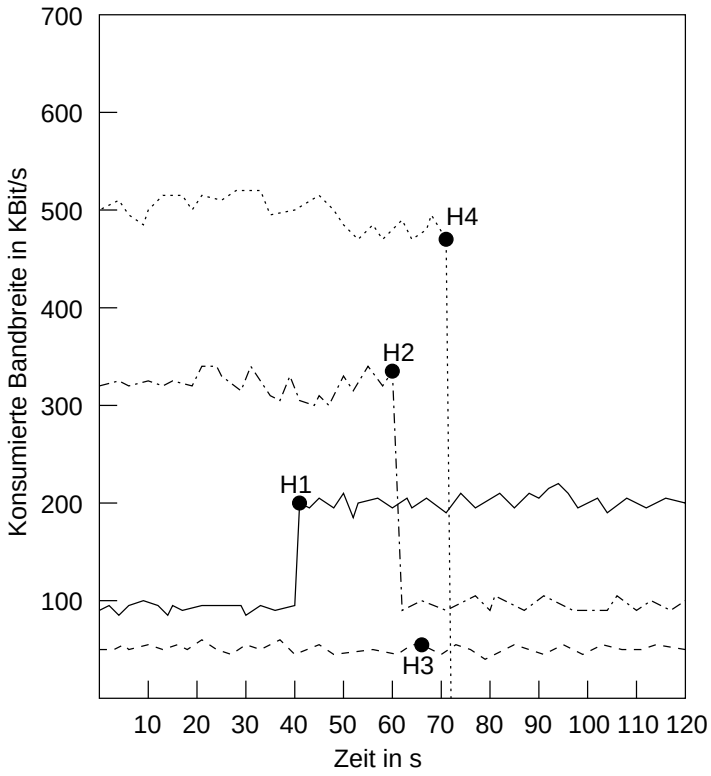


Abbildung 7: Testergebnisse des zweiten Versuchs zu Soft-State

Im folgenden Beispiel soll der Unterschied zwischen den verschiedenen Adaptionspolitiken aufgezeigt werden. Abbildung 7 zeigt, wieviel Bandbreite von den vier mobilen Geräten M1-M4 zu verschiedenen Zeitpunkten beansprucht wird. Zu beachten ist, daß in diesem Beispiel die Adaptionen nicht aufgrund der regelmäßigen Aktualisierungsbefehle der mobilen Geräte vorgenommen werden. Vielmehr soll hier das Adaptionsverhalten der Geräte beim Zellenwechsel direkt aufgezeigt werden. Die Geräte befinden sich am Anfang des Versuchs alle noch in verschiedenen Zellen, wechseln dann aber alle in die gleiche Mobilfunkzelle. M1 wechselt zuerst (Zeitpunkt H1) und beansprucht sofort einen Teil der in der neuen Zelle zur Verfügung stehenden Bandbreite für sich. Die Adaptionspolitk von M1 sieht Skalierungen des eigenen Bandbreitenkonsums nur beim Zellenwechsel vor. Da M1 nicht mehr wechselt, ändert sich die ihm zugewiesene Bandbreite auch nicht mehr. Zum Zeitpunkt H2 verläßt M2 seine alte Zelle und betritt die neue. Es muß dabei einen Großteil seiner vorher beanspruchten Bandbreite aufgeben und empfängt in der neuen Zelle nur noch eine Baselayer. Die Adaptionspolitik von M3 ist so ausgelegt, daß sich M3 niemals an Bandbreitenveränderungen anpasst. Nach dem betreten der neuen Funkzelle zum Zeitpunkt H3 behält M3 also auch seine bisherige Bandbreite.

Zum Zeitpunkt H4 versucht M4, in die neue Zelle zu wechseln, wird aber aufgrund mangelnder Ressourcen zurückgewiesen. Hier zeigen sich die Stärken, aber auch die Problematiken der Möglichkeit, die Adaptionspolitik pro Anwendung selbst festlegen zu können. Hätte Gerät M1 eine andere Adaptionspolitik gewählt, hätte es seine Enhancementlayer aufgeben können und so Gerät M4 den Eintritt in die neue Zelle ermöglicht . Andererseits können sich so Anwendungen, die auf ein gewisses höheres Bandbreitenniveau angewiesen sind, selbst vor ständigen Störungen schützen.

6 Fazit

Obwohl der normale MOBIWARE-Code ohne Verbesserungen gravierende Performanceprobleme aufwies, ist es den Entwicklern nicht zuletzt aufgrund ausgiebiger Tests gelungen, mit Hilfe einiger Verbesserungen eine stabil funktionierende Plattform für den Einsatz mobiler Rechner zu schaffen. Dabei sind vor allem auch die erheblichen Leistungsgewinne durch Einsatz von Flow-Bundling zu beachten, obwohl es in der Literatur auch durchaus Beispiele für schnellere Zellenwechsel gibt. Das Original-Paper verweist auf eine leider nicht näher angegebene Quelle von Partho, in der ein Zellenwechsel mit zehn beteiligten Datenströmen in 125 ms erreicht worden sein soll. Dieser Ansatz benutze allerdings reinen ATM-Code, so daß ihm im Gegensatz zu MOBIWARE die Möglichkeit fehlt, schnell eigene Objekte nach Bedarf zu programmieren und im System zu verwenden.

Es wurde in dieser Zusammenfassung ebenfalls auf die Vor- aber auch Nachteile verwiesen, die ein Konzept zur fairen Aufteilung der Ressourcen auf mehrere mobile Einheiten mit sich bringt. Besonders interessant ist dabei, daß die verwendeten Objekte verteilt auf jedem beliebigem Rechner im Netzwerk ausgeführt werden können und dann dort ihre Arbeit versehen. Eines der wichtigsten Konzepte von MOBIWARE ist jedoch die Bereitstellung von Grunddiensten, die eine einfache Anpassung an die Bedürfnisse künftiger verteilter Anwendungen in Mobilnetzen ermöglichen.

Es sei hier aber auch auf eventuell versteckte Probleme hingewiesen, welche leider in der Orginalliteratur nicht erwähnt werden. Insbesondere bei der Verwendung aktiver Filter ist Vorsicht geboten. Es ist unklar, welche Sicherheitsmechanismen MOBIWARE beinhaltet, die z.B. verhindern, daß ein aktiver Filter Daten verfälscht. Ebenso wichtig wäre es, sicherzustellen, daß MOBIWARE-interne Managementoperationen nicht unbeabsichtigt in die Arbeitsweise fremder Objekte eingreifen können. So muß beispielsweise ausgeschlossen werden, daß durch die Operationen eines mobilen Gerätes die Objekte eines anderen mobilen Gerätes neu konfiguriert werden. Ein weiterer möglicher Problempunkt ist die Skalierbarkeit von MOBIWARE. Es wird z.B. nicht näher erläutert, wie verhindert werden soll, daß das Netzwerk durch die Verwendung zu vieler Objekte und den dadurch anfallenden Verwaltungsaufwand ineffizient wird. Insbesondere die Tatsache, daß bei MOBIWARE momentan im gesamten Netz nur ein MobilityAgent-Objekt eingesetzt wird, könnte zu erheblichen Leistungseinbußen führen. Dieses Objekt spielt ja, wie bereits gezeigt, eine entscheidende Rolle bei Zellenwechsel- und Soft-State-Vorgängen. Unklar ist auch, nach welchen Kriterien abgerechnet werden soll. Wird nach Bandbreite auf der Funkstrecke bezahlt? Oder wird nach Bandbreite im ATM-Netz abgerechnet? Oder wird eventuell sogar eine ganz andere Abrechnungsmethode verwendet? Die Abrechnung nach konsumierter Funkstreckenbandbreite hätte den Vorteil, daß der Nutzer eines mobilen Gerätes nur für die Bandbreite bezahlt, die er auch wirklich nutzen kann. Andererseits macht es der Einsatz aktiver Medienfilter durchaus möglich, umfangreiche Datenmengen durch das ATM-Netzwerk zu transportieren und diese erst im Zugangspunkt auf eine geringere Datenmenge zu reduzieren. Die hohe Bandbreitennutzung im ATM-Netz würde sich dann aber in der Abrechnung nicht niederschlagen. Ein solches Verhalten würde die Einbeziehung der konsumierten Bandbreite im festverkabelten Netzwerk notwendig machen.

Zuletzt sei hier auch noch auf die tatsächlich bestehende Problematik der verantwortungslosen Anwendung möglicher Adaptionspolitiken hingewiesen, die ausfühlich in Abschnitt 5.1 behandelt wurde.

Literatur

[ACKL98a] O. Angin, A.T. Campbell, M.E. Kounavis und R. R.-F. Liao. Open Programmable Mobile Networks: Design, Implementartion and Evaluation. In *Proceedings of the Eighth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'98)*, Cambridge, England, Juli 1998.

[ACKL98b] O. Angin, A.T. Campbell, M.E. Kounavis und R.R.-F. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *IEEE Personal Communications* 5(4), August 1998, S. 32–43.

[ALLM96] C. Adam, A.A. Lazar, K.-S. Lim und F. Marconcini. xbind: The User's Manual. CTR Technical Report 452-96-16, Center for Telecommunications Research, Columbia University, New York, Juni 1996.

[BaAK97] A. Balachandran, A.T.Campbell und M.E. Kounavis. Active Filters: Delivering Scaled Media to Mobile Devices. In *Proceedings of the Seventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97)*, St. Louis, U.S.A., Mai 1997.

[BiCa98] G. Bianchi und A. T. Campbell. A Programmable MAC. In *IEEE International Conference on Universal Personal Communicatons (ICUP'98)*, Florence, Italy, Oktober 1998.

Entwurf verteilter mobiler Anwendungen mit ROVER

Ralf Hammer

Kurzfassung

Anwendungen, die auch in mobilen Umgebungen eine zuverlässige Leistung liefern sollen, müssen auf solche Umgebungen angepaßt werden. Das *ROVER*-Toolkit liefert dafür Programmier- und Kommunikationsabstraktionen, die es einer Anwendung erlauben, sowohl mit dem Benutzer als auch mit der *ROVER*-Laufzeitumgebung zu interagieren, um eine erhöhte Verfügbarkeit, Nebenläufigkeit, effiziente Betriebsmittelverwaltung, Fehlertoleranz und dynamische Anpassung an die aktuelle Umgebung zu erreichen. Tests an Applikationen, die nach *ROVER* portiert wurden, haben gezeigt, daß bei langsamer und nur gelegentlich vorhandener Verbindung zum Netz die Leistung der Benutzerschnittstelle stark erhöht und der Datenverkehr auf dem Netz dramatisch reduziert werden konnte.

1 Einleitung

In einer mobilen Umgebung existieren Einschränkungen, die in einer stationären Umgebung nicht vorkommen. Anwendungen, die für solche stationären Umgebungen geschrieben wurden, weisen auf einer mobilen Umgebung meist enorme Leistungsschwächen auf. Im folgenden werden diese Einschränkungen und die daraus resultierenden Probleme herkömmlicher Anwendungen skizziert.

1.1 Probleme bei Mobilrechnern

Die bei Mobilrechnern auftretenden Probleme beziehen sich sowohl auf deren Leistung als auch auf die Leistung der Kommunikationsverbindungen zur Außenwelt. Zu den Einschränkungen der Kommunikationsverbindung zählen u.a. die *begrenzte Bandbreite*, *hohe Verzögerungszeiten* aber auch *häufiger Verbindungsverlust*. Mobile Rechner weisen im Gegensatz zu ihren stationären Pendants meist *geringere Ressourcen* auf, zu denen die CPU-Leistung, die Grafikkarte und der Speicherausbau gehören. Zudem können sich beispielsweise bei einer Docking Station die *Ressourcen* (Co-Prozessor, Festplatten, größeres Display, Grafikprozessor) *dynamisch ändern*. Nicht zuletzt sind Mobilrechner *unzuverlässiger*, da sie aufgrund ihrer Mobilität Stürzen, Verlust und Diebstahl ausgesetzt sind. *Leistungsschwache Akkus*, deren Energie bereits nach wenigen Stunden aufgebraucht ist, stellen ebenfalls ein nicht zu vernachlässigendes Problem dar.

1.2 Probleme herkömmlicher Anwendungen

Für stationäre Rechner geschriebene Programme weisen meist eine schlechte Leistung auf, wenn sie auf mobilen Rechnern mit den oben erwähnten Einschränkungen konfrontiert werden. Sie zeichnen sich durch eine *langsame GUI* (Graphical User Interface) aus, die gerade bei sehr langsamen Netzen regelrecht "hängt". Außerdem erzeugen sie oft eine *unnötige Netzlast*, wenn beispielsweise nach einem Netzausfall Daten erneut geladen werden müssen, die vor dem Ausfall schon fast vollständig geladen wurden.

1.3 Zielsetzungen von ROVER

Das ROVER-System versucht sich nun genau dieser Probleme anzunehmen. ROVER wird seit 1995 am MIT (Massachussets Institute of Technology) entwickelt. Bei ROVER handelt es sich um ein *Toolkit*, das ein *konsistetes Framework in Form einer API* (Application Programming Interface) bietet, mit dessen Hilfe Anwendungsentwickler Programme für mobile Rechner entwickeln können. Zielgruppe von ROVER sind vor allen Dingen Rechner, die über sehr langsame Kommunikationsverbindungen verfügen. Das Toolkit unterstützt dabei die Entwicklung von Applikationen mit den folgenden Eigenschaften:

- Hohe Verfügbarkeit auch bei Netzausfall
- Effiziente Betriebsmittelverwaltung (Netzbandbreite, CPU-Leistung, Energie)
- Fehlertoleranz (aber dennoch gute Leistung)
- Adaption an sich verändernde Rahmenbedingungen
- Dynamischer Lastausgleich zwischen Client und Server
- Transparenz gegenüber der Mobilität
- Einbeziehung in Netzwerkfragen möglich

ROVER ermöglicht es durch diese Mobilitätstransparenz, für stationäre Rechner geschriebene Programme unverändert zu übernehmen. Dies entspricht auch den gegenwärtigen Bemühungen in anderen Bereichen, den Entwickler von unnötig erscheinenden Details zu entlasten. Beispiele hierfür sind *verteilte Datenbanken* (oder allgemeiner *verteilte Systeme*) mit ihrer Orts- und Netztransparenz oder aber die Programmiersprache *JAVA* mit ihrer Plattformunabhängigkeit. Allerdings versuchen die Entwickler von ROVER, die Entwicklung von sich ihrer Mobilität bewußten Applikationen (*mobile aware*) zu unterstützen, damit der Programmierer Expertenwissen einbringen kann, um die Leistung seiner Anwendung zu optimieren, was andernfalls nicht möglich ist. Die Entwickler von ROVER vertreten die Philosophie, daß dem Programmierer so viel Freiheit wie möglich beim Entwickeln überlassen werden sollte.

2 Überblick

ROVER bietet ein einheitliches verteiltes objektbasiertes System, das auf einer Client/Server-Architektur beruht. Um Daten zwischen Client und Server austauschen zu können, werden dem Anwendungsentwickler die Techniken *RDO (Relocatable Dynamic Object)* und *QRPC (Queued Remote Procedure Call)* zur Verfügung gestellt. Außerdem bietet ROVER noch ein ereignisgesteuertes System zur Abfrage des Status an. Diese grundlegenden Techniken werden in diesem Abschnitt näher beschrieben.

2.1 RDO (Relocatable Dynamic Object)

Dabei handelt es sich um ein *Objekt*, das Programmcode und Daten kapselt. Dieses Objekt besitzt eine *wohldefinierte Schnittstelle*, über die das Objekt abgefragt und verändert werden kann. Es stellt im Sinne der objektorientierten Programmierung ein Exemplar einer Klasse dar. Ein RDO sollte daher eine logische Einheit bilden, d.h. der Programmcode sollte mit den Daten assoziiert sein. Ein jedes RDO besitzt einen *Homeserver*, auf dem die Primärkopie liegt. Dieses RDO kann nun dynamisch vom Homeserver zum Client oder umgekehrt übertragen

werden, wobei ein RDO über den *URN (Universal Resource Name)* referenziert wird. Um bei einem Update-Konflikt durch mehrere Clients die Konsistenz wiederherstellen zu können, kann ein RDO mit zusätzlichen Methoden zur Konflikterkennung und -behebung ausgerüstet werden (siehe Abschnitt 5.4). Hier manifestiert sich wieder die Philosophie des ROVER-Teams, dem Anwendungsentwickler so viel Freiheiten wie möglich zu lassen.

2.2 QRPC (Queued Remote Procedure Call)

Ein normaler *synchroner RPC (Remote Procedure Call)* ruft auf einem entfernten Rechner eine Methode auf, die dort ausgeführt wird und danach eventuell Rückgabewerte an den Aufrufenden schickt. Erst danach kann der Aufrufende seine Arbeit fortsetzen; der (synchrone) RPC *blockiert* also das aufrufende Programm. Eine fehlende Verbindung zum Netz führt zum Fehlschlagen des RPC. Bei einem *asynchronen RPC* versucht man nun, dieses Blockieren zu umgehen, indem man einen eigenen Thread (parallel ablaufendes Programm) startet, der im Hintergrund diesen RPC (wieder synchron) durchführt, während allerdings das aufrufende Programm weiterläuft. Das Ende des RPC wird dem Aufrufenden durch Callback-Mechanismen mitgeteilt. Bei ROVER werden die RPCs in einer Logdatei protokolliert, die durch einen eigenen Thread bearbeitet wird. Hierdurch ist es nun möglich, auch bei fehlender Netzverbindung einen RPC zu senden, da er sich in eine *Warteschlange* einreiht. Im Unterschied zu normalen RPCs können bei QRPCs Anfragen und Antworten auf *unterschiedlichen Kanälen* gesendet werden, wobei auch *SMTP (!!!)* als Transportprotokoll verwendet werden kann. Um den Overhead des Verbindungsaufbaus zu umgehen, können auch mehrere RPCs zu einem zusammengefaßt werden, so daß nur noch ein einzelner RPC übertragen werden muß (*batching*). Durch *Kompression* der RDOs kann eine weitere Reduzierung des Kommunikationsaufkommens erreicht werden.

2.3 Ereignisse

ROVER überwacht ständig den Status des Netzwerks, der Hardwareressourcen und seine eigenen Aktivitäten. Der Anwendungsentwickler kann diesen Status durch Abfragen (*polling*) ermitteln oder *Callback*-Routinen registrieren, mit deren Hilfe er automatisch über Änderungen informiert wird. Zu den mitprotokollierten Ereignissen zählen:

- Erstellen und Löschen von ROVER-Clients
- Zustand des Netzwerkes (verbunden / nicht verbunden)
- Qualität des Netzwerkdienstes (Kosten, Verzögerung, Bandbreite)
- Warteschlange anstehender QRPCs
- Zustand der RDOs
  - vom Server angefordert
  - beim Client eingetroffen
  - vom Client modifiziert
  - Änderungen an Server übermittelt
  - Änderungen vom Server akzeptiert

3 Architektur

Die Architektur von ROVER ist in Abbildung 1 zu sehen. Es handelt sich dabei um eine Drei-Schichten-Architektur, die aus fünf Komponenten besteht, deren Aufgaben im folgenden näher erläutert werden.

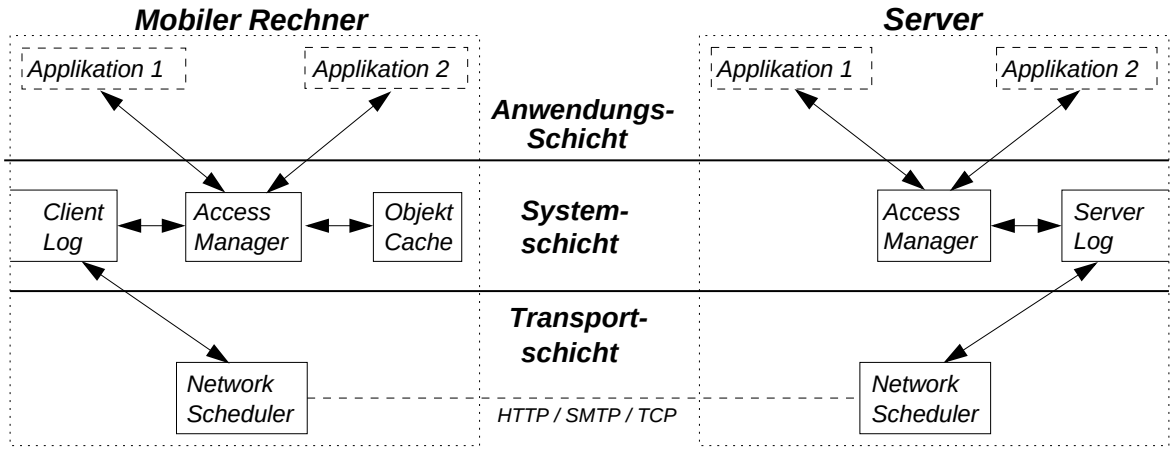


Abbildung 1: Architektur von ROVER

3.1 Access-Manager

Wie aus der Abbildung 1 ersichtlich ist, fällt dem Access-Manager eine zentrale Rolle zu. Client-Applikationen können Anfragen nach RDOs stellen, die der Access-Manager entweder mit Hilfe des Caches befriedigen kann, oder indem er die Anfrage in das Client-Log schreibt, das durch den Network-Scheduler asynchron bearbeitet wird. Konnte eine Anfrage nach einem RDO nicht sofort befriedigt werden, so erhält der Client bei Eintreffen des RDOs eine Nachricht (*Callback*). Führt ein Client Änderungen an einer lokalen Kopie eines RDOs durch, so protokolliert der Access-Manager dies mit, in dem er einen entsprechenden Eintrag im Client-Log tätig. Desweiteren kann eine Applikation zur Leistungssteigerung den Access-Manager anweisen, Objekte im voraus vom Server zu besorgen (*prefetch*). Dies geschieht über priorisierte Listen der Objekte, die man beispielsweise durch das Benutzerverhalten ermitteln kann.

3.2 Objekt-Cache

Auf dem Client existieren zwei Caches, ein *globaler Cache* im Adreßraum des Access-Manager, damit mehrere lokale Anwendungen RDOs gemeinsam benutzen können. Außerdem besitzt jede lokale Anwendung aus Effizienzgründen einen eigenen *lokalen Cache*, da eine lokale Applikation über Interprozeßkommunikation mit dem Access-Manager kommunizieren muß. Die Caches speichern den *letzten bekannten dauerhaften Zustand* eines RDOs und den *durch lokale Applikationen veränderten Zustand.* Als dauerhaften Zustand eines RDOs wird der Zustand der Primärkopie auf dem Homeserver angesehen (dieser kann also nur vom Server auf den neuesten Stand gebracht werden). Zur Konsistenzsicherung (relativ zur Primärkopie) werden mehrere Möglichkeiten angeboten, die man für ein RDO einstellen kann.

- nicht cachebar
- unveränderlich

- vor Benutzung überprüfen
- nur überprüfen, falls Verbindung zum Server besteht
- falls ein Verfallsdatum abgelaufen ist, muß eine neue Kopie besorgt werden
- Callback-Mechanismen

Um eine lokale Kopie eines RDO auf den neuesten Stand zu bringen, existieren ebenfalls mehrere Möglichkeiten.

- neue Kopie vom Homeserver beziehen
- nur durch andere Clients verursachte Änderungen vom Server beziehen
- nur durch lokale Applikationen verursachte Änderungen beziehen

3.3 Client-Log

Wenn ein Client eine Änderung an einem Objekt vornimmt, wird diese an der lokalen Kopie durchgeführt und zusätzlich ein entsprechender QRPC Eintrag in das Client-Log aufgenommen, um diesen bei Gelegenheit dem Server zu schicken, damit dieser ebenfalls die Änderung vornehmen kann. Um bei einem Konflikt, der beim Update eines Objektes beim Server durch mehrere Clients entstehen kann, flexibler reagieren zu können, werden im Client-Log auch die zugehörigen Methoden, die diese Änderungen verursachen, mitprotokolliert. Für jede Änderungsoperation werden noch applikationsspezifische Informationen abgelegt, die in Konfliktfällen ebenfalls dem Benutzer präsentiert werden können. Das dadurch entstehende schnelle Anwachsen des Logs wird vermindert, indem es den Applikationen ermöglicht wird, Prozeduren in den Access-Manager herunterzuladen, um direkt die eigenen Logbucheinträge zu manipulieren.

3.4 Network-Scheduler

Dieser bearbeitet im Hintergrund das Client-Log mit den QRPC Einträgen. Er gruppiert zusammengehörige Operationen, um sie als eine Einheit an den Server zu senden (*batching*). Dieses Gruppieren war im ersten Prototyp von ROVER nicht enthalten; es wurde erst später eingeführt (siehe [JoTK97]). *Umordnungen* können aufgrund von Konsistenzerfordernissen der Session oder aufgrund von applikationsspezifischen Prioritäten stattfinden. Diese Konsistenzerfordernisse und Prioritäten beeinflussen auch die *Wahl des Transportprotokolls* (SMTP oder TCP). Der Network-Scheduler entscheidet, welches Kommunikationsinterface geöffnet wird und was darüber gesendet werden soll. Dabei muß ein Kompromiß zwischen Qualität des Dienstes und verfügbarer Netzwerkleistung gefunden werden. Im Fehlerfall muß ein QRPC neu gesendet werden.

3.5 Server-Log

Dieses Server-Log wurde ebenfalls erst später in das ROVER-System eingebaut (siehe [JoKa97]).

3.5.1 Protokollieren eingehender QRPCs

Der Server protokolliert eingehende QRPCs, erst danach wird die Bestätigung gesendet. Dieses Protokollieren behebt den Verlust der QRPCs bei einem Server-Crash, d.h. der Client muß einen QRPC nicht erneut schicken, sofern er bereits eine Bestätigung erhalten hat. Beim Start der Ausführung eines QRPCs wird ein *'start-of-execution'*-Eintrag, nach der Ausführung ein *'completion'-Eintrag* in die Protokoll-Datei geschrieben. Anhand dieser zwei Einträge kann der Zustand eines QRPCs ermittelt werden.

- nicht ausgeführt
- nicht vollständig ausgeführt
- vollständig ausgeführt

Diese Vorgehensweise ist vergleichbar mit dem Protokollieren von Transaktionen in Datenbanken, bei dem *BOT (Begin Of Transaction)* und *EOT (End Of Transaction)* Einträge in die Protokolldatei geschrieben werden, um im Fehlerfall Transaktionen wiederholen zu können.

3.5.2 Sichern des Anwendungszustandes

Der Zustand der Server-Applikation wird durch persistente Variablen gespeichert. Im Programm werden Variablen, deren Inhalt es lohnt gesichert zu werden, um nach einem Server-Crash nicht erneut berechnet werden zu müssen, durch die Anweisung *Rover_ stable Variablenname [Startwert]* als persistente Variable deklariert. Änderungen an diesen Variablen werden durch die Trace-Funktion von Tcl überwacht und in die Protokolldatei geschrieben. Außerdem können mit der Anweisung *Rover_ setRecoveryProc Prozedurname* TCL-Recovery-Prozeduren deklariert werden, die nach einem Server-Crash einen geeigneten Zustand zum Wiederaufsetzen herstellen.

3.5.3 Sichern der Antwort eines QRPC

Schließlich wird auch die Antwort auf einen QRPC vor dem Senden zum Client protokolliert, falls eine erneute Erzeugung der Antwort zu teuer ist. Dies lohnt sich aber nur bei kleinen, rechenintensiven Antworten.

3.6 Beispielablauf einer Import-Anfrage

Um das Zusammenwirken einiger in diesem Abschnitt beschriebener Komponenten zu verdeutlichen, ist in der Abbildung 2 ein möglicher Ablauf einer Import-Anfrage einer Applikation nach einem RDO dargestellt.

4 Implementation

ROVER läuft sowohl auf Laptops unter Linux, DECstation unter Ultrix als auch auf SPARCstation unter SunOS. Der Server-Anteil von ROVER kann entweder als *CGI-Applikation* unter NCSA's httpd Server oder als einzelner TCP/IP-Server ausgeführt werden. Der Client-Anteil basiert auf der *Web Common Library* von CERN für HTTP-Unterstützung. Als Netzverbindungen werden sowohl Ethernet, WaveLan, ISDN als auch herkömmliche Telefon- und Funkverbindungen verwendet.

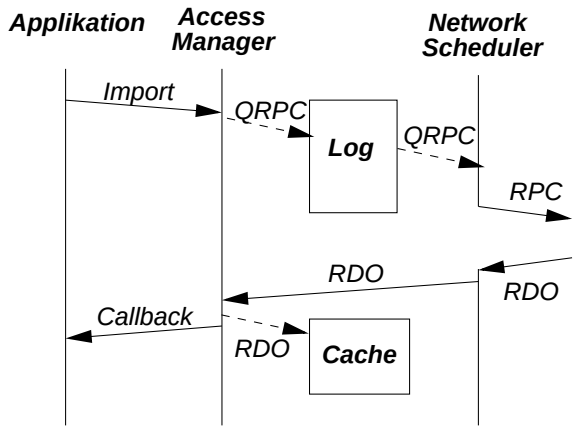


Abbildung 2: Ablauf einer Import-Anfrage

4.1 Applikationsschicht

ROVER Applikationen bestehen aus *Tcl/Tk-Skripten* und *binären Applikationen*. Diese Skripte werden durch einen Tcl/Tk-Interpreter verarbeitet, der eine C-Erweiterung besitzt, um RDOs zu unterstützten, und der mit der ROVER-Bibliothek gelinkt wird. Die binären Applikationen sind direkt mit den Kommunikationfunktionen der ROVER-Bibliothek verbunden.

4.2 Systemschicht

Jeder Client läuft in einem eigenen Adreßraum ab und kommuniziert mit dem lokalen Access-Manager über *IPC (Interprozeßkommunikation)*. Fehler in Clients wirken sich daher nicht auf andere Clients oder auf den Access-Manager aus. Der Access-Manager ist multithreaded ausgelegt, wobei einige dieser Threads im Hintergrund den Cache aufräumen. Die *Protokolldateien* wurden als *normale UNIX-Dateien* angelegt. Eine Datei speichert dabei das Inhaltsverzeichnis des Caches, und für jedes darin enthaltene Objekt existiert je eine weitere Datei mit dem Inhalt des Objekts und zusätzlichen Attributen wie Zeitpunkt und Kosten des Importierens und Zeitpunkt der letzten Benutzung. Aus Effizienzgründen ist das Inhaltsverzeichnis zusätzlich im Speicher vorhanden, wobei nur dieses immer auf dem neuesten Stand gehalten wird. Das Verzeichnis auf der Festplatte wird nur gelegentlich und im Hintergrund in einen konsistenten Zustand gebracht.

4.3 Transportschicht

Diese Schicht ist ihrerseits in zwei Schichten eingeteilt. Die *obere Schicht* wandelt die URNs (Universal Resource Name) in eine HTTP-Adresse um, die die Anfrage an eine RDO-Methode einschließlich einer Authentifizierung enthält. Die *untere Schicht* besteht aus dem Network-Scheduler und den Kommunikationsprotokollen. Durch Objektidentifikatoren kann hier sichergestellt werden, daß eine Anfrage höchstens einmal bearbeitet wird. Der Network-Scheduler fragt bei Erhalt eines QRPCs den Access-Manager, ob andere Clients auch QRPCs zum selben Server schicken wollen. Falls ein Client mehrere Anfragen schickt, werden sie automatisch zusammengefaßt. Der Network-Scheduler komprimiert den Kopf von Anfragen und - sofern keine applikationsspezifische Komprimierung vorhanden ist - auch die Nutzdaten. Die Transportschicht unterstützt verbindungsorientierte Protokolle wie HTTP über TCP/IP als auch verbindungslose Protokolle wie SMTP oder andere.

5 Entwurf mobiler Anwendungen

ROVER bietet Applikationen eine *Client/Server-Architektur*. Ein Teil der Applikation läuft auf dem Client-Rechner (meist ein mobiler Rechner) und ein weiterer Teil auf einem (stationären) Server. Beide Teile müssen mit der ROVER-Bibliothek gelinkt werden. Ein Anwendungsentwickler, der ROVER verwenden will, muß einige Entwurfsentscheidungen treffen, die im folgenden skizziert werden.

5.1 Mobilitätstransparenz versus Mobilitätsbewußtsein

Zuerst muß er entscheiden, ob er ein gegenüber der Mobilität transparentes oder ein in Mobilitätsfragen involviertes Programm erstellen beziehungsweise ein bestehendes anpassen will. Entschließt er sich für die Transparenz, so kann er bestehende Programme meist unverändert übernehemen, oder recht einfach ein neues erstellen, da er dabei von allen Netzwerkproblemen entlastet wird. Ist der Entwickler eher am zweiten Fall interessiert, so hat er die Möglichkeit, eine bessere Leistung aus dem Programm herauszuholen.

5.2 Entwurf der RDOs

Alle von einer Applikation benötigten Daten samt Methoden, die das Abfragen und Verändern der Daten ermöglichen, müssen in einem RDO (wie in Abschnitt 2.1 beschrieben) gekapselt werden. Dies gilt auch für Rückgabewerte von QRPCs. Die Granularität, mit der Daten in RDOs gekapselt werden, ist dabei ein wichtiger Parameter. Zu kleine RDOs erzeugen einen größeren Overhead beim Senden, zu große erzeugen dagegen bei Verlust ein erhöhtes Verkehrsaufkommen durch nochmaliges Senden. Außerdem ist der Prefetch-Mechanismus bei großen RDOs ineffizient. Mögliche Kandidaten für ein RDO sind beispielsweise *GUI-Code* oder *Berechnungen*.

5.3 Client/Server-Aufteilung

Ein weiterer wichtiger Parameter ist die Aufteilung der RDOs zwischen Client und Server. Wird beispielsweise das RDO mit der GUI auf dem Client ausgeführt, so reduziert sich das Verkehrsaufkommen zum Server drastisch, da die Größe des GUI-Codes im Vergleich zu den Updates der Grafikoberfläche, die der Code produziert, gering ist. Diese Aufteilung erhöht daher die angestrebte hohe Verfügbarkeit der Applikation. Genau dies wird auch bei JAVA-Applets erreicht, indem sie auf den Client heruntergeladen und dort ausgeführt werden. Ebenso könnten rechenintensive RDOs auf dem meist leistungsstärkeren Server ausgeführt werden, um die CPU des Mobilrechners zu entlasten. Falls eine große Datenmenge zum Client übertragen wird, von der ein großer Teil ausgefiltert werden muß, so wäre es sinvoller, ein mit diesem Filter versehenes RDO zum Server zu schicken, um bereits dort die Daten auszufiltern. Die Client/Server-Aufteilung sollte dynamisch geschehen, so daß sich das Programm an Veränderungen der verfügbaren Ressourcen anpassen kann.

5.4 Konsistenz

Wenn mehrere Clients, die auf verschiedenen Mobilrechnern laufen, Änderungen an RDOs vornehmen, dann kann es beim Server zu *Update-Konflikten* kommen. Einige der Konflikte können automatisch gelöst werden, andere hingegen nicht. Daher sollte der Anwendungsentwickler seine RDOs mit *Methoden zur Konflikterkennung und -behebung* ausstatten, da er

dadurch Expertenwissen verwenden kann, das nur er an dieser Stelle besitzt. Falls die Konflikte dennoch nicht aufgelöst werden können, besteht noch die Möglichkeit, den Benutzer in diesen Prozeß zu integrieren, indem er mit detaillierten Informationen über den Konflikt versorgt wird.

5.5 Benutzermitteilungen

Benutzermitteilungen können wie im obigen Fall der *Konfliktauflösung* dienen. Dem Benutzer können aber auch *allgemeine Informationen* über die Ereignisse, die (wie in Abschnitt 2.3 beschrieben) durch ROVER mitprotokolliert werden, präsentiert werden. So kann beispielsweise der Zustand eines RDOs auf der Grafikoberfläche durch unterschiedliche Farben dargestellt werden.

5.6 Optimierungen

Die Übertragung von RDOs zwischen Client und Server kann durch Ausstattung der RDOs mit Methoden zur *Komprimierung* der Daten verbessert werden. Außerdem kann das Programm dem Access-Manager priorisierte Listen mit Objekten übergeben, die bei Gelegenheit im Hintergrund vom Server angefordert werden können (*prefetch*).

6 Beispiele

Um die in den vorangegangenen Abschnitten erläuterten Techniken zu verdeutlichen, werden in diesem Abschnitt einige Beispielprogramme, die von dem ROVER-Team erstellt bzw. modifiziert wurden, vorgestellt. Zuerst werden jedoch noch einige Bibliotheksfunktionen von ROVER beschrieben, die von der Sprache Tcl aufgrufen werden können.

6.1 Bibliotheksfunktionen für den Server

Tcl-Funktion	*Beschreibung*
Rover_DbOpenRead	Öffnet eine Datenbank zum Lesen
Rover_DbClose	Schließt die Datenbank
Rover_AddLog	Fügt einem RDO-Log einen Eintrag hinzu
Rover_SetRecoveryProc	Setzt die Prozedur zur Fehlerbehebung
Rover_stable	Deklariert eine Variable als persistent
Rover_GetAccessTime	Liefert den Zeitpunkt der letzten Benutzung eines RDO

Tabelle 1: Bibliotheksfunktionen für den Server

Die genaue Syntax einiger in Tabelle 1 aufgeführter Funktionen wird im folgenden beschrieben.

- *Rover_ DBOpenRead(Dateiname, Modus)*
- *Rover_ AddLog(Objektname, Zeit, Status, Operation)*. Status kann dabei einen der beiden Werte *commit* oder *abort* annehmen. Operation gibt den Namen der Prozedur an, die auf das Objekt angewendet wurde.
- *Rover_ SetRecoveryProc Prozedurname*
- *Rover_ stable Variablenname [Startwert]*
- *Rover_ SetAccessTime(Objektname, Zeit)*

6.2 Bibliotheksfunktionen für den Client

Tcl-Funktion	*Beschreibung*
Rover_NewSession	Stellt eine Verbindung zum lokalen Access-Manager her
Rover_Import	Importiert ein RDO (vom lokalen Cache oder vom Server)
Rover_Export	Exportiert ein RDO (zum Homeserver)
Rover_PromiseClaim	Wartet auf den Wert eines Promise
Rover_IsTentative	Testet, ob lokales RDO mit Server RDO übereinstimmt
Rover_GetDV	Liefert den Abhängigkeitsvektor eines RDOs
Rover_QRPC	Führt einen nicht blockierenden RPC aus
Rover_RPC	Führt einen blockierenden RPC aus
Rover_Shutdown	Beendet eine lokale Applikation

Tabelle 2: Bibliotheksfunktionen für den Client

- Rover_NewSession liefert einen Session-ID, der von allen anderen Funktionen als Parameter benötigt wird. Folgende Parameter müssen dieser Funktion übergeben werden.
  1. service (Angeforderter Dienst)
  2. consistency (Grad der geforderten Konsistenz)
  3. exclusivity (Möglichkeit, Daten zwischen Anwendungen zu teilen)
  4. user
  5. password
- *Rover_ Import(sessionID, URL, Callback)* liefert einen Promise-ID. Die URL gibt dabei das angeforderte RDO an und die Callback-Methode wird (falls vorhanden) bei Eintreffen des RDO aufgerufen.
- *Rover_ PromiseClaim(promiseID)* kann verwendet werden, um den Wert eines durch Rover_Import/Export erhaltenen Promise abzufragen.

6.3 Mobilitätstransparente Anwendung

Als ein Beispiel für eine mobilitätstransparente Anwendung wird im folgenden der Web-Browser *Rover Mosaic* vorgestellt, dessen Architektur in Abbildung 3 zu sehen ist. Dabei handelt es sich um das unverändert übernommene Programm *Mosaic* und um zuätzlichen Komponenten, die diesen Web-Browser auf Mobilrechner, die über sehr langsame Verbindungen zum Netz verfügen, anpassen.

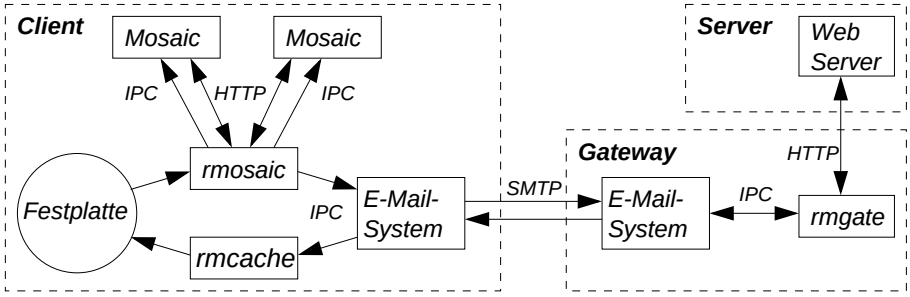


Abbildung 3: Architektur von Rover Mosaic

6.3.1 Ziele von Rover Mosaic

Diese Anpassung hatte das Ziel, die *Verfügbarkeit der GUI* zu erhöhen, indem dem Benutzer erlaubt wird, Links anzuklicken, die dann im Hintergrund bearbeitet werden, während im Vordergrund weitergearbeitet werden kann (*click-ahead*). Es sollte also auch möglich sein, mehrere Links in Hintergrund bearbeiten zu lassen und bei Eintreffen der entsprechenden Information den Benutzer geeignet darüber in Kenntnis zu setzen. Außerdem sollte die verfügbare Bandbreite des Netzes optimal ausgenutzt werden, indem automatisch Informationen auf Vorrat vom Server angefordert werden, falls das Netz nicht anderweitig verwendet wird (*prefetching*).

6.3.2 Ablauf einer Anfrage

Zum besseren Verständnis wird in diesem Abschnitt der Ablauf einer Anfrage näher erläutert. Der Benutzer bedient das Programm Mosaic wie gewohnt. Klickt er auf einen Link, so wird diese Anfrage an den HTTP-Proxy *rmosaic* gesendet. Dieser beantwortet die Anfrage entweder mit Hilfe des Caches, falls die entsprechende Information darin enthalten ist, andernfalls wird eine leere Antwort gesendet, so daß Mosaic seine Arbeit fortsetzen kann. War die Information nicht im Cache vorhanden, so wird zuätzlich diese Anfrage als eMail an einen Gateway-Server gesendet. Das Programm *rmgate* bearbeitet auf dem Gateway diese ankommenden Anfragen, indem sie als "echte" HTTP-Anfrage an den Server weitergereicht werden. Die Antwort auf diese HTTP-Anfrage wird dann wieder durch *rmgate* über eMail an den Client zurückgeschickt. Auf dem Client-Rechner werden ankommende eMail-Nachrichten durch das Programm *rmcache* bearbeitet, das die Daten in einem Cache auf der Festplatte speichert. Außerdem informiert es den Proxy *rmosaic* über die eingetroffenen Daten. Der Proxy *rmosaic* wiederum erzeugt eine dynamische HTML-Seite, in der Informationen über den Zustand angeforderter Daten angezeigt wird. Diese Seite wird an die zweite Instanz des Mosaic gesendet, der diese Cache-Liste (wie in Abbildung 4 zu sehen) anzeigt.

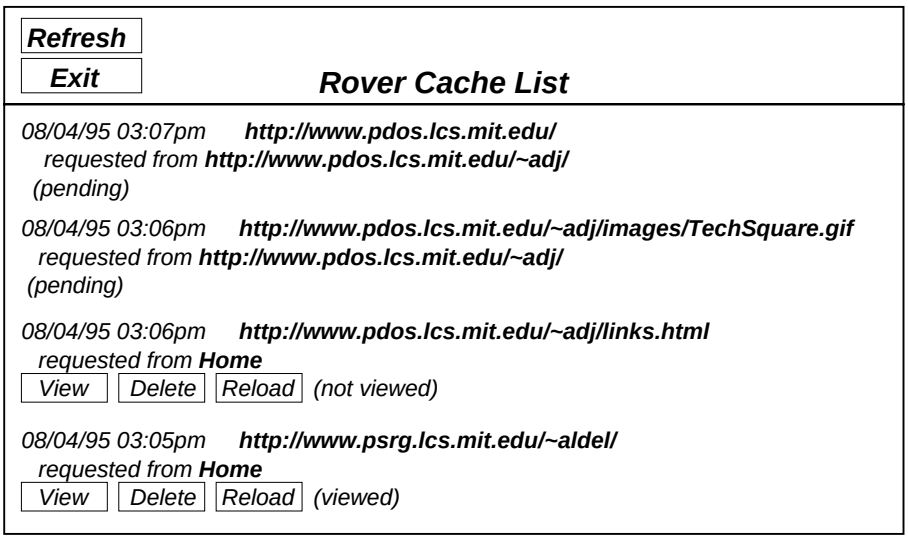


Abbildung 4: Cache-Liste von Rover Mosaic

6.4 Mobilitätsbewußte Anwendung

In diesem Abschnitt wird das Programm *Webcal* vorgestellt. Dabei handelt es sich um den nach ROVER portierten Terminplaner *Ical* von Sanjay Ghemawat.

6.4.1 Grundlagen von Ical

Da es sich bei Ical um einen gewöhnlichen Terminplaner mit *X-Oberfläche* handelt, werden die entsprechenden Funktionen wie Hinzufügen, Löschen und Editieren von Terminen, die einen Start- und einen Endzeitpunkt besitzen, angeboten. Die Termine können dabei in einer *Monats-/Wochen- oder Tagesansicht* dargestellt werden. Ical speichert Kalender, die aus einer Menge von Terminen bestehen, in einer einzelnen Datei, die aufgrund seltener Entfernung von Terminen durch den Benutzer meist stark anwächst. Durch diese Dateiorientierung stellt eine Datei auch zugleich die Einheit der Konsistenz dar. Versuchen zwei Benutzer, den selben Kalender zu verändern, so führt dies zu einem Konflikt, der durch die Zeitmarke der entsprechenden Datei erkannt wird. Um diese Konflikte zu minimieren, wird die Datei häufig gespeichert.

6.4.2 Probleme von Ical auf Mobilrechnern

Wird *Ical* nun auf einer mobilen Umgebung gestartet, so zeigen sich (besonders bei langsamen Verbindungen zum Server) enorme Schwächen. Ändert der Benutzer beispielsweise nur einen Termin in einem größeren Kalender, so muß durch die Dateiorientierung der komplette Kalender gespeichert werden, was über eine langsame Verbindung extrem lange dauern kann oder bei Verbindungsverlust gar unmöglich ist. Außerdem laufen die Versionen des Kalenders auf dem Mobilrechner und auf dem Server bei längerem Verbindungsverlust auseinander, so daß ein Update-Konflikt bei erneuter Verbindung zum Server wahrscheinlicher wird. Wird gar das X-Interface über das Netz gestartet, so läuft es recht langsam ab und ist bei Verbindungsverlust unbenutzbar (siehe auch Abschnitt 1.1 und 1.2).

6.4.3 Verbesserungen in Webcal

Webcal versucht diesen Problemen durch drei Neuerungen gegenüber *Ical* Rechnung zu tragen.

- *Feinkörnige Objektorientierung statt grobkörniger Dateiorientierung.* Jeder Termin wird in einem eigenen RDO gekapselt und kann daher unabhängig von den anderen importiert, exportiert und zwischengespeichert werden. Dies führt zu einer *Verminderung der Konfliktwahrscheinlichkeit*, da ein Konflikt nur durch Veränderung ein und desselben Termins durch mehrere Benutzer entsteht. Tritt ein Konflikt auf (der u.U. erst nach mehreren Stunden oder Tagen erkannt wird, da erst dann die Verbindung mit dem Server wiederhergestellt werden kann), so kann der konfliktverursachende Termin dem Benutzer präsentiert werden, damit er diesen manuell auflöst. Ein weiterer Vorteil der feinkörnigen RDOs besteht in der *Verminderung der Netzlast*, da bei einem Update nur noch die Daten gesendet werden müssen, die sich wirklich verändert haben.

- *Die Aufgaben werden auf Client und Server verteilt.* Die GUI läuft beim Client und weist daher eine hohe Verfügbarkeit auf. Der Server kümmert sich um die Datenspeicherung und um Serialisierung konkurrierender Updates.

- *Benutzerschnittstelle, die die Mobilität berücksichtigt.* Die GUI wurde im Vergleich zu Ical dahingehend erweitert, daß der Zustand der RDOs, der über das ROVER-System abgefragt werden kann, durch unterschiedliche Farben dem Benutzer deutlich gemacht wird.

7 Vergleich/Ausblick

Im wesentlichen wurden die Ziele, die sich das ROVER-Team gesteckt hat (vgl. hierzu Abschnitt 1.3), erreicht. Tests einiger nach ROVER portierter Programme (u.a. die in Abschnitt 6 beschriebenen) haben noch einige Erkenntnisse gebracht.

7.1 Testergebnisse

- Einfache Portierung von Programmen, die ein Dateisystem-Modell zur Speicherung der Daten benutzen (nur etwa 10% des Codes waren zu ändern)
- Einige Anwendungen konnten unverändert übernommen werden
- Zwischenspeichern von RDOs reduziert den Bedarf an Bandbreite
- Nachrichtenverkehr ist proportional zu den Änderungen an RDOs
- Leistung eines QRPC ist auch bei Protokollieren auf die Festplatte und bei Benutzung des SMTP-Protokolls akzeptabel, da die Kosten bei langsamen Verbindungen viel höher sind
- Persistente Variablen auf der Server-Seite stellen einen einfachen Mechanismus dar, um fehlertolerante Applikationen zu unterstützen
- Die Leistung von ROVER Anwendungen ist über schnelle Verbindungen nur unwesentlich langsamer als die ursprüngliche Anwendung
- Die Leistung von ROVER Anwendungen ist besonders über langsame Verbindungen deutlich höher als bei herkömmlichen Anwendungen

7.2 Vergleich zu anderen Entwürfen

ROVER stellt die erste implementierte Architektur dar, die sowohl mobilitätstransparente System-Proxies als auch sich ihrer Mobiliät bewußte Applikationen unterstützt. Außerdem unterstützt kein anderes System RDO oder QRPC in der Form, wie dies bei ROVER der Fall ist. Das *Coda Projekt* bietet verteilte Dienste für mobile Clients, wobei versucht wird, Mobilität vor der Anwendung zu verstecken. Dabei werden alle Änderungen, die eine Anwendung während eines Netzausfalls auf dem Dateisystem vornimmt, protokolliert und bei erneuter Verbindung zum Netz abgespielt. Zur Konfliktauflösung werden auf der Ebene der Verzeichnisse automatische Mechanismen angeboten, während auf der Dateiebene applikationsspezifische Mechanismen unterstützt werden. Für nicht auflösbare Konflikte wird ein manuelles Reparaturkit zur Verfügung gestellt. Das *Bayou Projekt* besitzt eine Architektur, die es mobilen Clients erlaubt, Daten gemeinsam zu verwenden. Dieses Projekt spezialisiert sich im Gegensatz zu *Coda* mehr auf mobilitätsbewußte Applikationen. Es unterstützt nichtpersistente Zustände der Daten, Garantieleistungen für Sessions und ist daher mit ROVER in diesen Punkten sehr gut zu vergleichen. Allerdings besitzt dieses Projekt keine RDOs oder QRPCs. RDOs lassen sich noch am Besten mit *Agenten* oder mit der Programmiersprache *JAVA* vergleichen. Der Hauptunterschied zu ROVER besteht allerdings darin, daß ROVER RDOs mit einer wohldefinierten, objektbasierten Laufzeitumgebung anbietet, die eine einheitlich Namensgebung, applikationsspezifische Replikationsmodelle und nicht zuletzt QRPC zur Verfügung stellt.

7.3 Ausblick

Für die Zukunft sind noch weitere Verbesserungen des ROVER-Toolkits vorgesehen. Da es sich noch um Prototypen handelt, die als Testumgebung für verschiedene Ideen und zum besseren Auffinden von Fehlern ausgelegt sind, bieten sich noch an einigen Stellen Leistungssteigerungen an. Zu den geplanten Änderungen gehören u.a. die Möglichkeit, den Anwendungsentwickler in den Batching-Prozeß miteinzubeziehen, und die Weiterentwicklung der nach ROVER portierten Programme.

Literatur

[deLe95] A.F. deLespinasse. Rover mosaic: E-mail Communication for a Full-Function Web Browser. Diplomarbeit, Massachusetts Institute of Technology, Juni 1995.

[JdTG+95] A.D. Joseph, A.F. deLespinasse, J.A. Tauber, D.K. Gifford und M.F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Copper Mountain, CO, USA, Dezember 1995.

[JoKa97] A.D. Joseph und M.F. Kaashoeck. Building Reliable Mobile-Aware Applications Using the Rover Toolkit. *Wireless Networks* 3(5), Oktober 1997, S. 405–419.

[JoTK97] A.D. Joseph, J.A. Tauber und M.F. Kaashoeck. Mobile Computing with the Rover Toolkit. *IEEE Transaction on Computers: Special Issues on Mobile Computing* 46(3), März 1997, S. 337–352.

[JTGC+] A.D. Joseph, J.A. Tauber, D.K. Gifford, G.M. Candea und F. Kaashoek. Rover Mobile Application Toolkit. http://www.rover.lcs.mit.edu/.

[Taub96] J.A. Tauber. Issues in Building Mobile-Aware Applications with the Rover Toolkit. Diplomarbeit, Massachusetts Institute of Technology, Juni 1996.

Vernetzung über GSM - Die Mowgli-Architektur

Stefan Geretschläger

Kurzfassung

Die „Mowgli"-Kommunikationsarchitektur versucht, eine Brücke zwischen der Datenübertragung über Funkstrecke und der Datenübertragung innerhalb verkabelter Netze zu schlagen. Konkreter: Mobile Rechner sollen durch die Mowgli-Architektur mit möglichst guten technischen Leistungen, mit möglichst hoher Fehlertoleranz und mit möglichst hohem Benutzerkomfort über GSM an verkabelte Netze angebunden werden. Die Architektur passt sich an existierende Anwendungen (z. B. E-Mail-, Web-Klienten) sowie an bestehende Festnetz-Architekturen, die auf TCP/IP basieren, an. Neue, speziell für die Mowgli-Architektur entwickelte (kommunizierende) Anwendungen bieten neue Funktionalitäten und mehr Bedienungskomfort.

1 Einleitung/Motivation

Portable Computer, vor allem PCs, und Mobiltelefonie (z. B. GSM) sind in der heutigen Zeit weit verbreitet. Zusammengenommen ergeben diese beiden Systeme eine neue Plattform für ubiquitären Zugriff auf Datenbestände und Rechendienste in fest verdrahteten Netzen, insbesondere dem Internet.

Die hierbei zu überbrückende drahtlose Verbindungsstrecke bringt jedoch unangenehme Effekte für den Benutzer mit sich, falls sie bei der Übertragung der Daten nicht besonders behandelt wird.

1.1 Das Mowgli-Projekt

Seit Mitte der 90er Jahre gibt es an der University of Helsinki ein Projekt namens „Mowgli" , welches Architekturen untersucht, entwirft und testet, deren Zweck es ist, mobile Rechner möglichst „gut" über zellulare GSM-Telefonsysteme an fest verdrahtete Netzwerke (wie z. B. das Internet) anzubinden. Man versucht hierbei, die oben angesprochenen Probleme zu beseitigen oder wenigstens abzumildern. „Mowgli" ist ein Akronym und steht für „Mobile Office Workstations using GSM Links". Das Projekt wird mitfinanziert von Nokia, DEC und Telecom Finland.

1.2 Formulierung der Problemstellung

Das GSM-Telefonsystem ermöglicht ein komfortables Telefonieren, sofern sich der Benutzer in einer Gegend mit ausreichender Netzabdeckung befindet, und stellt somit auch mobilen Rechnern neben der eigentlichen Übertragung von Daten einiges an Funktionalität zur Verfügung. GSM sorgt so z. B. dafür, dass die Anbindung eines mobilen Rechners an das Festnetz auch dann dauerhaft und transparent gewährleistet ist, wenn sich dieser von einer Funkzelle in die nächste bewegt („hand-over").

Werden nun in einer bestehenden, im Festnetz gut funktionierenden Kommunikationsarchitektur die Kommunikationsdienste der unteren beiden Schichten („physical layer“ und „data link layer“, vgl. ISO/OSI-Schichtenarchitektur) durch die GSM-Technologie ersetzt, so entsteht eine neue Architektur, die jedoch nicht den Bedürfnissen und Anforderungen gerecht wird, die der Benutzer eines mobilen Rechners zweifelsohne hat. So ist es z. B. nicht möglich, gebräuchliche E-Mail- oder WWW-Klienten auf einem mobilen Rechner flüssig und reibungslos einzusetzen.

Eine von vielen möglichen Ursachen wäre beispielsweise: Ein mobiler Rechner bewegt sich auch in Gegenden hinein, die vom zellularen Telefonsystem nicht abgedeckt werden. Und das GSM-Telefonsystem kann oft auch aus anderen Gründen ein Mindestmaß an Übertragungsqualität nicht zur Verfügung stellen, z. B. aufgrund von Funkschatten, die Gebäude werfen, oder aufgrund von längeren Tunnels.

Bestehende und gebräuchliche Kommunikationsdienste (z. B. TCP/IP/PPP) sowie kommunizierende Anwendungen (z. B. E-Mail bzw. WWW) verhalten sich in solchen typischen Situationen nicht angemessen. Sie sind für Verbindungen über schnelle und zuverlässige Netzwerke entwickelt worden. Mit den knappen Ressourcen (Übertragungsleistung, etc.) gehen sie nicht besonders sparsam um. Und statt Ausnahmen und Fehler angemessen zu behandeln, muss sich der Benutzer an der Benutzeroberfläche mit ihnen auseinandersetzen.

1.3 Anforderungen an eine neue Architektur

„Mowgli“ hat sich nun zum Ziel gesetzt, mittels einer geeigneten Architektur und den darin integrierten Konzepten, die „Lücken“ zwischen der Funktionalität eines GSM-Systems und den Wünschen und Bedürfnissen der Benutzer zu schließen. Diese Lücken werden nun im Folgenden aufgeführt:

- Die Folgen schlechter Sende- bzw. Empfangsbedingungen sollen abgemildert bzw. minimiert werden. Die Fehlertoleranz wird dadurch verbessert.
- Im „normalen“ Betrieb soll die nicht sehr hohe Übertragungsleistung (Bandbreite) einer GSM-Verbindung besser und ökonomischer ausgenutzt werden.
- Idealerweise sollen sich kommunizierende Anwendungen auf einem mobilen Rechner möglichst plausibel, intuitiv, flüssig und ähnlich wie auf einem Rechner im Festnetz bedienen lassen. Dies ist u. a. auch deshalb wünschenswert, weil sich das Spektrum der Benutzer immer weniger auf Computer-Experten beschränkt [KALR95].

Damit die „Mowgli-Architektur“ in der Praxis zum Einsatz kommen kann, muss sie notwendigerweise auch noch den folgenden technischen Anforderungen nachkommen:

- Die Zusammenarbeit mit bereits existierenden, auf TCP/IP beruhenden Kommunikationsarchitekturen im Festnetz muss gewährleistet werden. Dies schließt insbesondere bereits existierende TCP/IP-Protokoll-Software mit ein.
- Die neue Mowgli-Architektur muss auf den mobilen Rechnern den schon existierenden, verbreiteten Kommunikationsanwendungen die Funktionalitäten der gewohnten Dienste anbieten.
- Es muss ein API („application programming interface“) zur Verfügung gestellt werden, welches die Entwicklung von neuen Klienten-Anwendungen ermöglicht [KALR95].

2 Die generellen, konkreten Schwierigkeiten

Die Übertragung von Daten über ein fest verdrahtetes Netz ist heutzutage schnell, effizient und zuverlässig. Nicht so einfach ist es jedoch, auch einen Benutzer zufriedenzustellen, der seinen Standort ständig ändert. Denn die drahtlose Übertragung größerer Datenmengen bringt generelle Schwierigkeiten mit sich.

Unter guten Sende- bzw. Empfangsbedingungen ist mit Anwendungen, die über eine GSM-Funkstrecke kommunizieren, noch ein annehmbares Arbeiten möglich, wenn auch die technischen Leistungen hierbei nicht so gut sind wie bei der ausschließlichen Verwendung eines festen Telefonnetzes [AKLR96].

In der Praxis ist die Übertragung von Daten über eine drahtlose Verbindungsstrecke jedoch sehr langsam und anfällig für Störungen. Im Vergleich zur „konventionellen“ Übertragung von Daten sind also deutliche Einbußen bei den technischen Leistungen und damit auch bei der Dienstgüte zu verzeichnen.

Die konkreten Probleme, die sich bei der Datenübertragung über GSM generell ergeben, sind nun hier im Folgenden aufgeführt:

- Der Durchsatz bei einer GSM-Verbindung („GSM data service“) beträgt nominal 9.600 bps. In der Praxis werden aber selbst unter guten und verlässlichen Bedingungen meist nicht mehr als 7.200 Bit/s erreicht [KKLR96].
- Der Aufbau einer GSM-Verbindung dauert relativ lange.
- GSM bietet nur verbindungsorientierte Dienste auf Basis einer Leitungsvermittlung an. Die drahtlose Übertragung von Daten ist daher für den Benutzer ein relativ teures Unterfangen. Falls der Benutzer nämlich eine bestehende GSM-Verbindung längere Zeit ungenutzt lässt (z. B. weil er bis zur nächsten interaktiven Eingabe mit Lesen beschäftigt ist), so ist dies unökonomisch.
- Die Nachrichtenlaufzeit (bzw. die Verzögerung) variiert abhängig von der Häufigkeit der Übertragungsfehler auf der drahtlosen Verbindungsstrecke sehr stark, da der leistungsfähige GSM-Fehlerkorrektur-Mechanismus (Radio Link Protocol) viel Zeit für sich in Anspruch nimmt. Unter ungünstigen Bedingungen kann die Nachrichtenlaufzeit gar mehr als eine Minute betragen.
- Die Anbindung des mobilen Computers über eine Funkstrecke an das fest verdrahtete Netz ist stark störungsanfällig. Bewegt sich der portable PC in ein vom Mobiltelefonie-Provider nicht abgedecktes Gebiet hinein, oder nimmt die Sende-/Empfangsqualität aufgrund anderer Störeinflüsse stark ab, so kann die Anbindung des mobilen Rechners an das Festnetz vollkommen unerwartet (zeitweilig oder dauerhaft) abreißen [AKLR96].

3 Nutzung von konventionellem TCP/IP

Die Ursachen dafür, dass das TCP/IP-Protokoll für die Übertragung von Daten über eine GSM-Verbindung nicht geeignet ist, sind vielfältig. Hier nun eine Aufzählung der wichtigsten Ursachen:

- Weder das TCP/IP-Protokoll noch zur Zeit existierende Anwendungen reagieren angemessen auf Bedingungen und Vorkommnisse, wie sie bei der drahtlosen Übertragung von Daten gang und gäbe sind. Daher führen auch kleine Störungen schon zu Fehlfunktionen, die der Benutzer auf der Anwendungsebene deutlich wahrnimmt und die den Benutzer bei der Verrichtung seiner Arbeiten behindern.

- IP, TCP, UDP und PPP benutzen umfangreiche Header, was viel Protokoll-Overhead beim Datenverkehr über die GSM-Verbindung zur Folge hat.

- Bei den genannten Protokollen verzögern bestätigte Dienstleistungen und somit verdoppelte Nachrichtenlaufzeiten („round trips“) häufig die Abarbeitung nachfolgender Operationen, denn die einfache Nachrichtenlaufzeit auf der Funkstrecke ist schon sehr lang und variiert stark (vgl. Abschnitt 2).

- Der Slow-Start-Algorithmus von Jacobsen zur Überlastungsvermeidung stellt ein prinzipielles Problem beim Einsatz von TCP auf Funkstrecken dar. Dem Algorithmus liegt die Annahme zugrunde, dass Timeouts durch Überlastung verursacht werden. Diese Annahme trifft jedoch bei der Datenübertragung über eine GSM-Verbindung nicht zu. Außerdem kommt der Slow-Start-Algorithmus unnötigerweise jedes Mal zum Einsatz, wenn aufgrund schlechter Bedingungen eine GSM-Verbindung wieder aufgebaut wird.

  Beim Einsatz von TCP auf einer drahtlosen Verbindungsstrecke werden Timeouts dadurch verursacht, dass Datenpakete auf der Funkstrecke verloren gehen, bzw. daduch, dass (alternativ) der vom GSM verwendete, sehr leistungsfähige Fehlerkorrekturmechanismus des „Radio-Link-Protokolls“ die Nachrichtenlaufzeiten stark verlängert.

  „Durch Verlangsamung der Übertragung (gemäß dem Slow-Start-Algorithmus) verschlimmert sich die Situation noch. Der richtige Ansatz ist hier, verlorene Pakete so schnell wie möglich erneut zu übertragen [Tane98].“

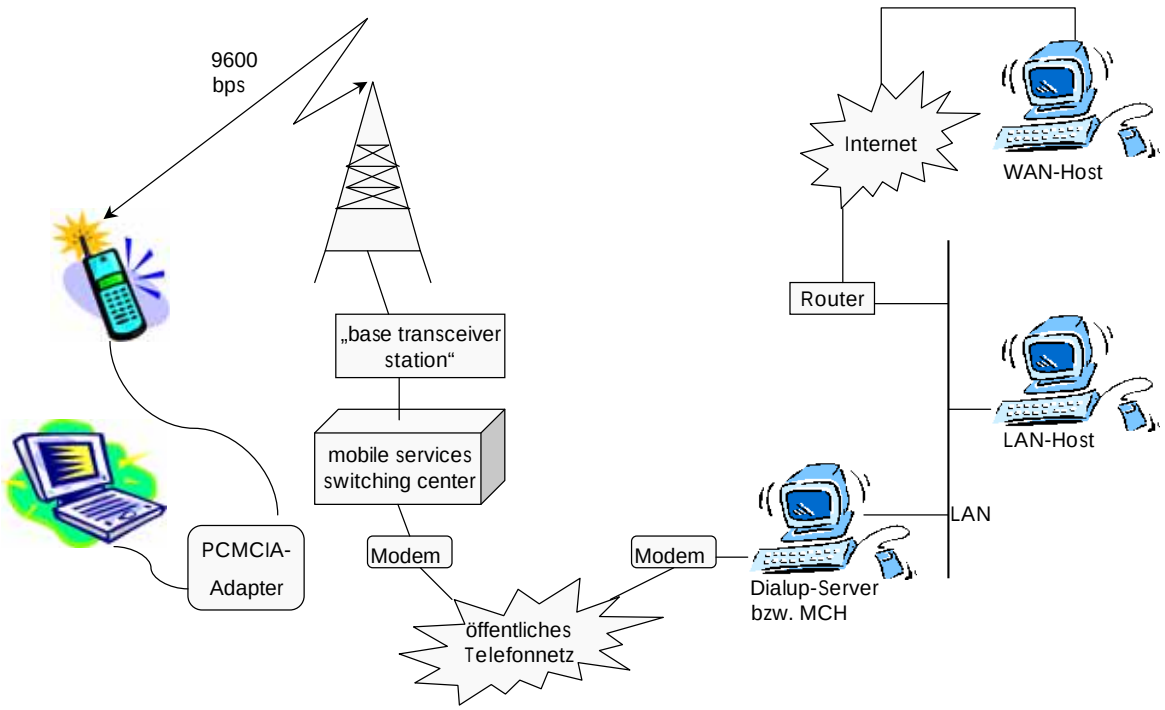


Abbildung 1: Der Weg vom mobilen Rechner zum Kommunikationspartner im Internet

4 Das grundlegende Paradigma der Mowgli-Architektur

Hauptgegenstand der Forschungsarbeiten im Mowgli-Projekt ist die „Mowgli-Architektur“. In ihr findet sich der folgende, wesentliche Ansatz wieder:

Die Datenübertragung über Funkstrecke und die Datenübertragng über das Festnetz werden unterschiedlich behandelt, nämlich dem zugrunde liegenden Medium jeweils möglichst gut angepasst. Die Strecke, die eine Ende-zu-Ende-Verbindung überbrückt (vgl. Transportschicht des ISO/OSI-Schichtenmodells), wird deshalb in zwei Teilstrecken aufgesplittet: erstens die Funkstrecke, die nur schwache technische Leistungen zur Verfügung stellen kann, und zweitens die restliche überbrückte Strecke im fest verdrahteten Netz [KKLR96]. Man erhält so zwei getrennte Kommunikations-Subsysteme.

Die grundlegende Idee der Mowgli-Architektur ist nun jedoch die „indirekte Interaktion“, die zwischen einer kommunizierenden Anwendung auf einem mobilen Rechner und einem Kommunikationspartner im Festnetz stattfindet [KALR95].

Wie in Abbildung 1 beispielhaft dargestellt befindet sich an der Schnittstelle zwischen der (die Funkstrecke beinhaltenden) Telefonverbindung und dem Internet der „mobile connection host“ (MCH). Für den mobilen Computer stellt er nach dem Aufbau einer GSM-Verbindung einen Einstiegspunkt für die Datenübertragung ins Internet dar [KKLR96].

Auf dem mobilen, portablen Rechner und auf dem MCH befindet sich nun der sogenannte „Mediator“, ein transparenter, verteilter und intelligenter Agent, der die gesondert zu behandelnde Funkstrecke sozusagen einklammert bzw. umschließt. Der Mediator besteht aus zwei Komponenten, dem Agenten auf dem mobilen Computer und dem Proxy auf dem MCH. Die beiden Komponenten bilden zusammen ein Team und befinden sich beide innerhalb der Ende-zu-Ende-Verbindung, die (auf Anwendungsebene) zwischen kommunizierender Anwendung des mobilen Rechners und Kommunikationspartner im Festnetz besteht. Das gewöhnliche Client-Server-Paradigma wird so durch ein Client-Mediator-Server-Paradigma, und das Peer-to-Peer-Paradigma durch ein Peer-to-Peer-through-Mediator-Paradigma ersetzt [KKLR96].

Sollen Daten zwischen einer Anwendung auf dem mobilen Computer und einem Kommunikationsparter irgendwo im Festnetz (z. B. dem Internet) ausgetauscht werden, so wird der Anwendung vom Mediator nach dem Stellvertreterprinzip die Rolle des Kommunikationspartners vorgegaukelt.

Auf der Seite des fest verdrahteten Netzes spielt der Mediator die Rolle der Anwendung, und der Kommunikationspartner kann nicht zwischen Mediator und Anwendung unterscheiden. Die Mowgli-Architektur passt sich so der bereits im Einsatz befindlichen TCP/IP-Software im Festnetz an und kommt damit einer der in Abschnitt 1.3 aufgeführten Anforderungen nach.

Ermöglicht wird dies, indem der MCH für jeden mobilen Rechner, der eine Verbindung zum MCH aufgebaut hat, ein „virtuelles Netzwerk-Interface“ einrichtet, über das Daten mit Kommunikationspartnern im Internet ausgetauscht werden können. Falls sich der MCH nicht im heimatlichen Intranet befindet, wird dabei in der Regel das „mobile IP“ angewandt, so dass der mobile Rechner für einen Kommunikationspartner im Festnetz nicht vom MCH zu unterscheiden ist.

5 Einblicke in die verschiedenen Schichten

Es folgt nun ein kurzer, einführender Überblick über die Schichten der Mowgli-Architektur:

- Die Datentransportschicht ist die unterste, grundlegende Schicht auf der drahtlosen Verbindungsstrecke zwischen mobilem Computer und MCH. Aufgabe dieser Schicht ist es, den zeitlichen Ablauf von in Auftrag gegebenen Datenübertragungen feinkörnig zu planen.
- Die Datentransferschicht ist über der Datentransportschicht angesiedelt. Ihre Dienste können optional in Anspruch genommen werden. Der Hauptzweck dieser Schicht besteht darin, zu übertragende Daten zu puffern und den zeitlichen Ablauf grob zu planen, falls mehrere Übertragungen in Auftrag gegeben wurden.
- Die Agent-Proxy-Schicht ist für anwendungsspezifische Optimierungen zuständig. Sie beinhaltet die oben eingeführten Agenten und Proxies und ist die oberste Schicht der Mowgli-Architektur, die die gesondert zu behandelnde Funkstrecke umschließt.
- Die kommunizierenden Anwendungen des mobilen Rechners greifen über das Mowgli Socket Interface auf die Dienste der Datentransportschicht, der Datentransferschicht und der Agent-Proxy-Schicht zu.

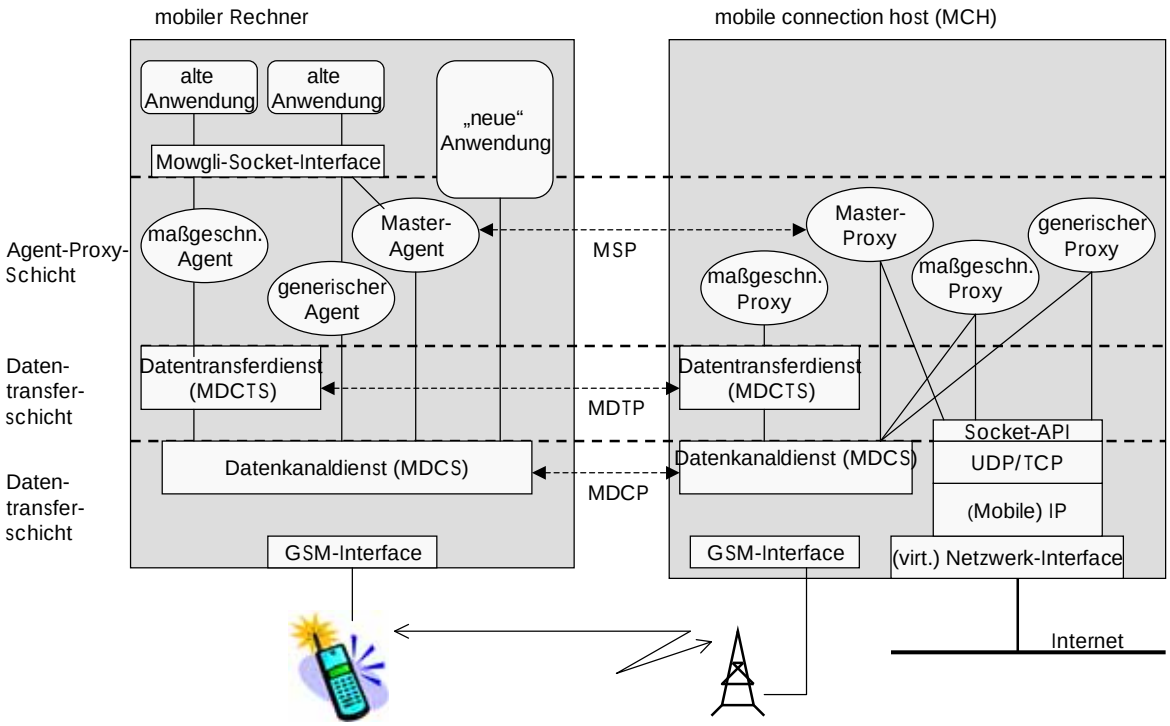


Abbildung 2: Zugriff auf die Dienste der Mowgli-Architektur über das Mowgli Socket Interface

Ein grafische Darstellung der Schichten zeigt Abbildung 2. Auf ihr kann man unter anderem auch erkennen, dass die horizontale Kommunikation jeder Schicht wie üblich über ein ihr eigenes Protokoll erfolgt.

Vergleicht man die Mowgli-Architektur mit der konventionellen Internet-Kommunikationsarchitektur (basierend auf TCP/IP), so sind von den neuen Architekturbestandteilen alle Schichten des ISO/OSI-Schichtenmodells betroffen, sowohl anwendungs- als auch transportorientierte Schichten. Dies unterscheidet die Mowgli-Architektur von vielen anderen Architekturen, die ähnliche Ziele verfolgen.

Die nächsten Teilabschnitte gehen nun sehr detailliert auf die einzelnen Schichten der Mowgli-Architektur und die zugehörigen Konzepte ein. Die unterste Schicht wird dabei im Folgenden als erste vorgestellt. Danach folgen – wie schon bei der Aufzählung der Schichten in diesem Kapitel – die höherliegenden Schichten („bottom up“). Die Reihenfolge entspricht dabei der Anordnung der Schichten in der Mowgli-Architektur.

5.1 Datentransportschicht

5.1.1 Grundlegende Begriffe

Von der Datentransportschicht wird der Datenkanaldienst (mowgli data channel service, MDCS) angeboten, dessen Aufgabe es ist, die standardisierten Dienste, die normalerweise von TCP/IP zur Verfügung gestellt werden, möglichst effizient und transparent zu ersetzen. Außerdem soll durch den MDCS im Vergleich zu TCP/IP die Fehlertoleranz verbessert werden. Die Datentransportschicht soll sich den speziellen Gegebenheiten des zellularen GSM-Telefonsystems also besser anpassen als eine herkömmliche auf TCP/IP basierende Kommunikationsarchitektur (vgl. Abschnitte 2 und 3).

Intern verwaltet die Datentransportschicht hierzu bis zu 256 bidirektionale (logische) Datenkanäle, durch die Datenpakete über die Funkstrecke geleitet werden. Die Flusssteuerung eines

Kanals ist unabhängig von der der anderen Kanäle. Und das gleichzeitige Durchführen von mehreren Übertragungsvorgängen (multiplexing) wird selbstverständlich unterstützt.

Es gibt zwei verschiedene Typen von Kanälen: „stream channels“ und „message channels“. Erstere stellen die Funktionalität des TCP zur Verfügung und letztere die Funktionalität des UDP.

5.1.2 Das Attributsystem der Datenkanäle

Ein Datenkanal hat (ähnlich wie DTOs in Abschnitt 5.2.3) Attribute, über die ein MDCS-Klient das Systemverhalten der Datentransportschicht bestimmen bzw. beeinflussen kann.

Es folgt nun eine Übersicht über die verschiedenen Kanalattribute und deren Bedeutungen [KKLR96]. Dabei ist stets das Folgende zu beachten: Wenn keine GSM-Verbindung besteht, so heisst das nicht, dass alle Kanäle geschlossen sein müssen. Diese beiden Sachverhalte sind vielmehr unabhängig voneinander zu sehen.

1. Die zwei Übertragungsrichtungen eines jeden Kanals haben jeweils ein Attribut „Priorität“, mit dessen Hilfe die feinkörnige zeitliche Abfolge beim gleichzeitigen Durchführen von mehreren Übertragungsvorgängen (multiplexing) beeinflusst werden kann. Außerdem wird die zur Verfügung stehende Bandbreite mit Hilfe der Werte dieses Attributs auf mehrere Kanäle aufgeteilt.

2. „GSM-Verbindungserlaubnis“: Dieses Attribut enscheidet darüber, was passiert, wenn momentan keine GSM-Verbindung besteht und ein Kanal von einem MDCS-Klienten zu übertragende Daten angeliefert bekommt. Es gibt drei mögliche Werte für dieses Attribut.

   (a) Der erste Wert „auto“ bewirkt, dass im oben beschriebenen Fall automatisch eine GSM-Verbindung aufgebaut wird.

   (b) Der zweite Wert „ask“ bewirkt, dass der Benutzer gefragt wird, ob er eine GSM-Verbindung aufgebaut haben möchte.

   (c) Der dritte Wert „use“ bewirkt, dass die zu übertragenden Daten zwischengespeichert werden. Denn Daten werden bei diesem Wert nur dann übertragen, wenn (durch die Übertragungswünsche anderer Kanäle) bereits eine GSM-Verbindung besteht.

3. Über das Attribut „Persistenz“ (Beharrlichkeit) kann ein MDCS-Klient einstellen, was passieren soll, wenn eine GSM-Verbindung unerwartet abbricht. Auch hier stehen mehrere Verhaltensweisen zur Auswahl:

   (a) Der erste Wert „when-data“ bewirkt, dass nur dann versucht wird, die GSM-Verbindung automatisch wiederherzustellen, wenn sich im Kanal noch zwischengespeicherte Daten befinden, die noch nicht über die Funkstrecke übertragen wurden.

   (b) Der zweite Wert „auto“ bewirkt, dass in jedem Fall (auch ohne noch zu übertragende Daten im Kanal) versucht wird, die aufgrund ungünstiger Bedingungen abgebrochene GSM-Verbindung wiederherzustellen, sofern der Kanal noch offen ist bzw. existiert.

   (c) Der dritte Wert „none“ bewirkt, dass die abgebrochene GSM-Verbindung nicht wiederhergestellt wird.

4. „Zustandstimer“: Über dieses Attribut kann ein MDCS-Klient einstellen, wie lange die in Abschnitt 5.1.3 erwähnten, zum „Wiederaufsetzen“ benötigten Zustandsinformationen erhalten bleiben, falls nach einem unerwarteten Verbindungsabbruch längere Zeit keine GSM-Verbindung wieder aufgebaut wird bzw. wieder aufgebaut werden kann. Falls dieser Timeout eintritt, so wird der Kanal geschlossen und alle zwischengespeicherten, noch zu übertragenden Daten werden (mitsamt den Zustandsinformationen) verworfen.

5. Das fünfte und letzte Attribut „Leerlaufzeit“ gibt an, wie lang in einem Datenkanal „Leerlauf“ andauern muss, bevor die Datentransportschicht eine bestehende GSM-Verbindung (geordnet) abbaut. Voraussetzung für das Trennen einer GSM-Verbindung ist jedoch, dass die „Leerlaufzeit“ anderer offenen Kanäle ebenso abgelaufen ist. Wird die GSM-Verbindung abgebaut, so bedeutet dies (noch einmal zur Erinnerung) nicht, dass die bestehenden Kanäle geschlossen werden. Zu einem späteren Zeitpunkt, wenn wieder zu übertragende Daten von einem MDCS-Klienten angeliefert werden, kann die Datentransportschicht (bei entsprechender Attributbelegung evtl. auch automatisch) wieder eine GSM-Verbindung aufbauen.

5.1.3 Die Konzepte des Datenkanalprotokolls

1. Über die Funkstrecke, die nur wenig Bandbreite zur Verfügung stellen kann, sollen nur die unbedingt erforderlichen Daten übertragen werden. Deshalb ist das Mowgli-Data-Channel-Protokoll (MDCP) ein sehr leichtgewichtiges Protokoll, das möglichst wenig „Protokoll-Overhead“ verursacht [KKLR96]:

   (a) Die Protokoll-Header der Daten- und Steuerungspakete werden beim MDCP so kurz wie möglich gehalten. Der Header eines Datenpakets umfasst nur ein bis drei Bytes, der eines Steuerungspaketes zwei bis 16 Bytes, im Mittel vier Bytes.

   Häufig vorkommende Situationen, bei denen ein Teil der vollständigen Header-Information entbehrlich ist, werden ausgenutzt. Beispielsweise kann der Kanal-Bezeichner weggelassen werden, wenn ein Paket an den gleichen Kanal wie das vorangegangene Paket adressiert wird.

   (b) Das Radio-Link-Protokoll stellt zwar einen starken Fehler-Korrektur-Mechanismus zur Verfügung, jedoch kann es vorkommen, dass andere Komponenten zwischen portablem Rechner und „mobile connection host“ Bitfehler verursachen. Für diesen Fall kann das MDCP zu einem „error-monitoring“-Modus umschalten, um Fehler zu erkennen, und den Daten Prüfsummen hinzufügen. Da dies aber nicht oft nötig ist, arbeitet das MDCP meist in einem „default“-Modus ohne Prüfsummen.

2. Die zur Flusssteuerung notwendigen „Acknowledgements“ und die daraus resultierenden „round-trips“ werden beim MDCP wegen der sehr langen Nachrichtenlaufzeiten auf der Funkstrecke möglichst sparsam eingesetzt. Stattdessen wird beim MDCP eine (im Vergleich zum TCP erweiterte) Variante des kreditbasierten Flusssteuerungsalgorithmus „sliding-windows“ verwendet, bei der außer dem Kredit, den der Empfänger dem Sender vergibt, auch noch die „neue Quota“ eine Rolle spielt.

   Unter der neuen Quota ist der Speicherplatz auf der Empfängerseite zu verstehen, der für weitere zu empfangende Daten reserviert werden kann. Die neue Quota kann dem Sender also im Rahmen eines neuen Kredites zusätzlich zur Verfügung gestellt werden. In der Regel wächst die neue Quota, wenn Daten an (im Schichtenmodell) weiter oben liegende Schichten weitergegeben werden.

   Ein neuer Kredit wird dem Sender jedoch nur dann eingeräumt, d. h. man schickt dem Sender nur dann ein „Acknowledgement“ zu, wenn der Kredit des Senders unter

Konzept	Nutzen
verbesserte Flusssteuerung	weniger bestätigte Dienstleistungen und weniger „round trips“
automatisches (Wieder-)Herstellen einer GSM-Verbindung	mehr Benutzerkomfort
bei vorübergehendem Verbindungsabbruch: „recovery“	höhere Fehlertoleranz
Prioritäten der Kanäle	kürzere Antwortzeiten bei Interaktivitäten
Vermeidung von Leerlauf	ökonomische Nutzung der GSM-Verbindung
Minimierung des Protokolloverheads	Engpass „Funkstrecke“ entschärft

Abbildung 3: Die Konzepte des Datenkanalprotokolls und ihr Nutzen

einen Schwellwert sinkt und die neue Quota über einen gewissen Schwellwert steigt. Die Einstellung der Schwellwerte ist maßgeblich für einen flüssigen Datenfluss und einen minimalen Overhead durch Acknowledgements [KKLR96].

3. Beim „Multiplexing“ kann die zeitliche Abfolge der Datenübertragung auf verschiedenen Kanälen durch den Einsatz von Prioritäten beeinflusst bzw. gesteuert werden.

   Wollen mehrere Datenkanäle gleichzeitig Daten übertragen, so wird die Bandbreite der GSM-Verbindung (zunächst) ausschließlich den Datenkanälen mit der höchsten der vorhandenen Prioritätsstufen zur Verfügung gestellt. Datenkanäle gleicher Prioritätsstufe teilen sich die Bandbreite fair nach dem Zeitscheibenverfahren.

   Läuft nun z. B. eine interaktive mit dem Festnetz kommunizierende Anwendung zeitgleich mit einer Datenübertragung, die längere Zeit im Hintergrund abläuft und die viel Bandbreite in Anspruch nimmt, so kann den von der interaktiven Anwendung stammenden, zu übertragenden Daten eine höhere Priorität zugewiesen werden. Durch den Einsatz von Prioritäten können so Antwortzeiten stark verkürzt werden.

   Ergänzend wäre noch zu bemerken, dass die höchste Priotät stets den Steuerungspaketen vorbehalten bleibt.

4. Falls eine GSM-Verbindung unerwartet und vorübergehend aufgrund schlechter Sende-/Empfangsbedingungen abbricht, setzt der MDCS effiziente, ausgefeilte Methoden ein, um auf einen Stand (wieder-)aufzusetzen, den man bei der Datenübertragung bereits erreicht hat („recovery“). Die Fehlertoleranz bei der Datenübertragung über die störanfällige Funkstrecke wird so spürbar verbessert. Der MDCS verwaltet deshalb zu jedem Kanal Zustandsinformationen während einer laufenden Datenübertragung.

   Im Falle eines unerwarteten Abbruchs der GSM-Verbindung kann der MDCS ein Signal oder eine Nachricht an die MDCS-Klienten abgeben, so dass diese in einem getrennten Modus (vgl. Abschnitt 5.3.2) weiterarbeiten können, sofern sie einen solchen besitzen.

   Bricht eine GSM-Verbindung unerwartet ab und sind die Attribute der bestehenden Kanäle entsprechend gesetzt, so kann das System versuchen, eine GSM-Verbindung automatisch (bzw. nach Rückfrage des Benutzers) wieder herzustellen [KKLR96].

Zusammendfassend stehen die Konzepte des Datenkanalprotokolls in Abbildung 3 noch einmal ihrem Nutzen gegenüber.

5.1.4 Das API der Datentransportschicht

Die Funktionalität der Datenkanal-Schicht wird höheren Schichten als „mowgli data channel service“ (MDCS) mittels eines APIs zur Verfügung gestellt. Dieses API ähnelt formal und in seiner Funktionalität – wie das Mowgli Socket Interface – (vgl. Abschnitt 5.4) dem „BSD Socket Interface“ [AKLR96].

Das API stellt den MDCS-Klienten folgende Operationen zur Verfügung [KKLR96]:

1. Operationen zum Erzeugen eines Kanals mit vorher festgelegten Attributen. Falls keine Attribute vorher festgelegt werden, werden Vorgabewerte verwendet.

2. Operationen zum Verändern von Attributen bereits erzeugter Kanäle, z. B. falls ein Klient plötzlich Daten zu verschicken hat, die wichtiger sind als vorangegange.

3. Operationen zum Senden von Daten in einen Kanal bzw. zum Empfangen von Daten aus einem Kanal.

4. Operationen zum Zurücksetzen eines Kanals (reset). Ruft ein MDCS-Klient eine solche Operation auf, werden die sich bereits im Kanal befindlichen, noch nicht über die Funkstrecke verschickten Daten verworfen.

5. Operationen, mit deren Hilfe der Benutzer des mobilen Rechners eine GSM-Verbindung aufbauen bzw. abbauen, also direkt beeinflussen kann.

6. Operationen zum Abrufen von Statistiken, die der MDCS automatisch verwaltet. So stellt der MDCS z. B. Informationen zur Qualität der GSM-Verbindung und zu deren Auslastung zur Verfügung.

Falls ein MDCS-Klient der Datentransportschicht Daten übergibt, und diese Daten werden aufgrund der momentanen Attributbelegungen der existierenden Kanäle nicht sofort übertragen, sondern zwischengespeichert, so besteht außerdem noch die Möglichkeit, dass der MDCS-Klient ein Signal oder eine Nachricht erhält, sobald die Übertragung zu einem späteren Zeitpunkt erfolgreich stattgefunden hat.

5.2 Datentransferschicht

5.2.1 Datentransferobjekte

In der Datentransferschicht steht der Begriff des Daten-Transfer-Objekts (DTO) im Mittelpunkt. Aus der Sicht des Benutzers ist ein DTO ein als Einheit zu behandelndes, elementares, zu sendendes oder zu empfangendes Informationspaket, das unabhängig von anderen DTOs behandelt werden kann. E-Mail-Nachrichten, zu übertragende Dateien oder Bilder innerhalb einer Web-Seite können beispielsweise als DTOs angesehen werden [AKLR96]. Mit anderen Worten: Ein DTO ist ein grundlegendes Element strukturierter Information [KALR95].

Die Funktionalität der Datentransfer-Schicht wird höheren Schichten über den „mowgli data transfer service“ (MDTS) und sein API zur Verfügung gestellt.

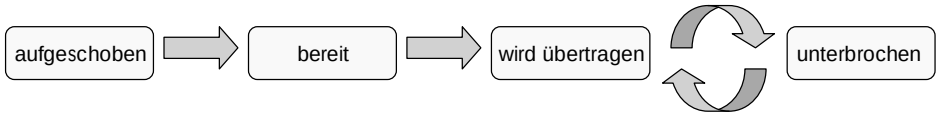


Abbildung 4: Der Lebenszyklus eines Datentransferobjekts

5.2.2 Die Zustände der Datentransferobjekte

Ein DTO befindet sich stets in genau einem von vier verschiedenen Zuständen.

1. „aufgeschoben“: Die Übertragung des DTO hat noch nicht begonnen und kann aufgrund gewisser Umstände zur Zeit auch noch nicht begonnen werden.
2. „bereit“: Die Übertragung des DTO hat noch nicht begonnen, kann aber zum aktuellen Zeitpunkt begonnen werden.
3. „wird übertragen“: Das DTO wird gerade übertragen.
4. „unterbrochen“: Die Übertragung des DTO wurde gestartet und ist aufgrund gewisser Umstände vorübergehend unterbrochen worden.

Der Lebenszyklus mit den möglichen Zustandsübergängen ist in Abbildung 4 dargestellt. Unter welchen Bedingungen die Zustandsübergänge stattfinden, ist im nächsten Abschnitt erklärt.

5.2.3 Das Attributsystem der Datentransferobjekte

Der Übertragungszeitpunkt eines DTO und auch das Systemverhalten während einer DTO-Übertragung wird durch die Attribute eines DTO gesteuert bzw. beeinflusst. Aufgrund dieser Attribute – man spricht sogar von einem System von Attributen – kann die Datentransferschicht selbstständig und unabhängig von höheren Schichten Entscheidungen darüber fällen, ob, wann, wie und in welcher Reihenfolge sie DTOs unter bestimmten Bedingungen überträgt.

Im Folgenden werden die Attribute eines DTO aufgeführt:

- Zeit: Dieses Attriut gibt den Zeitpunkt an, nachdem die Übertragung des DTO beginnen kann. Bei der Belegung dieses Attributs mit einem Wert können auch die unterschiedlichen Telefontarife eine Rolle spielen.
- GSM-Verbindungsqualität: Das Attribut ist ein Schwellwert. Die Bitübertragungsrate, die bei einer DTO-Übertragung zum aktuellen Zeitpunkt zu erwarten wäre, muss diesen Schwellwert übersteigen.
- Verbindungstyp: Es gibt die möglichen Werte „stark verbundener Modus“ und „schwach verbundener Modus“. Nimmt das Attribut den Wert „stark verbundener Modus“ an, so kann die DTO-Übertragung nur bei diesem Modus stattfinden. Beim Wert „schwach verbundener Modus“ ist eine DTO-Übertragung sowohl im „stark verbundenen Modus“ als auch „schwach verbundenen Modus“ möglich. Der „fast getrennte Modus“ und der „getrennte Modus“ bleiben hier außen vor, da hier keine DTOs übertragen werden können (vgl. Abschnitt 5.3.2).
- Warten: Es handelt sich um eine explizite Sperre, die vom Benutzer gesetzt oder gelöscht werden kann. Solange diese Sperre gesetzt ist, kann keine DTO-Übertragung stattfinden.

Die bisherigen Attribute werden auch Bedingungsattribute genannt und sind sozusagen „guards“ (Wächter) für die Übertragung der DTOs.

Ein DTO ist im Zustand „aufgeschoben“, wenn mindestens eine Bedingung gegen eine Übertragung spricht. Sobald kein Bedingungsattribut mehr gegen eine Übertragung spricht, geht der DTO-Zustand über in „bereit“. Dieser Zustandsübergang kann auch durch einen expliziten Benutzerbefehl erfolgen.

Ein weiterer „guard“ ist nicht spezifisch für ein einzelnes DTO, sondern erlaubt die Übertragung einer Reihe von DTOs erst dann, wenn genug Datenvolumen zusammengekommen ist, so dass dann die Bandbreite der zugrunde liegenden GSM-Verbindung ökonomisch genutzt werden kann.

Falls sich ein DTO im Zustand „wird übertragen“ befindet und „plötzlich“ ein Bedingungsattribut gegen eine Übertragung spricht, sei es, weil sich der Verbindungstyp bzw. die Verbindungsqualität geändert hat, oder sei es, weil der Benutzer ein Bedingungsattribut ändern möchte, so geht der DTO-Zustand über in „unterbrochen“.

Der Zustandsübergang von „unterbrochen“ nach „wird übertragen“ erfolgt wie der von „aufgeschoben“ nach „bereit“.

Bleibt der Zustandsübertrang von „aufgeschoben“ nach „bereit“. Hier spielt das nächste Attribut eine Rolle:

- Übertragungspriorität: Mit Hilfe dieses Attributs wird die Reihenfolge ermittelt, in der die DTOs den Kanälen des Datenkanaldienstes zur Übertragung übergeben werden. Falls ein MDTS-Klient ein DTO anliefert, dessen Priorität höher ist als die eines anderen DTO, das gerade übertragen wird, so wird das gerade angelieferte DTO sofort an einen passenden Kanal des Datenkanaldienstes weiterübergeben [KALR95]. Die Priorität eines DTO entspricht dabei stets der Priorität des korrespondierenden Kanals in der Datentransportschicht.

Die Datentransferschicht bestimmt so die grobe zeitliche Abfolge, in der die DTOs übertragen werden. Die Datentransportschicht führt dann im Gegensatz dazu eine feinkörnigere Planung durch und bestimmt, welchen Anteil an der verfügbaren Bandbreite die einzelnen Kanäle zugeteilt bekommen.

- Operation: Dieses Attribut gibt an, welche DTO-spezifische Operation der Sender des DTO vor der Übertragung aufruft. Der Empfänger des DTO ruft nach der Übertragung die korrespondierende Operation auf. Beispielsweise kann ein DTO vor dem Verschicken über die Funkstrecke komprimiert und danach wieder dekomprimiert werden.
- Benachrichtigung („notification“): Ist dieses Attribut entsprechend gesetzt, so wird der Benutzer nach Übertragung des DTO benachrichtigt (vgl. Abschnitt 5.1.4). Empfänger der Benachrichtigung ist normalerweise die „Anwendung zur Kommunikationskontrolle“ (vgl. Abschnitt 5.5) oder eine für die Mowgli-Architektur speziell entwickelte Anwendung.

Weitere Attribute eines DTO entsprechen den bereits erwähnten Attributen „GSM-Verbindungserlaubnis“, „Persistenz“ und „Zustandstimer“ eines (übertragenden) Datenkanals (vgl. Kanalattribute 2 bis 4 in Abschnitt 5.1.2, vgl. auch Abschnitt 5.2.5).

Und zuguterletzt besitzt ein DTO noch Attribute, mit deren Hilfe sich das Systemverhalten im Falle gewisser Konflikte steueren lässt. Ein solcher Konflikt liegt z. B. dann vor, falls eine Kontrolloperation des Benutzers direkt oder indirekt zur Zustandsänderung einer gerade in Übertragung befindlichen DTO führt.

5.2.4 Informationsflüsse, Gruppen und Warteschlangen

Bei vielen Anwendungen können zu übertragende Daten nicht als eine Folge von diskreten, voneinander unabhängig steuerbaren DTOs angesehen werden. Werden solch unstrukturierte Daten übertragen, so spricht man von einem Informationsfluss. Ein Informationsfluss stellen z. B. die Daten dar, die während einer Telnet-Sitzung übertragen werden, oder ein Strom von Multimediadaten.

Ein Informationsfluss kann nur zwei Zustände haben: „wird übertragen“ und „unterbrochen“. Ist ein Informationsfluss im Zustand „wird übertragen“, so werden die zugehörigen Daten sofort übertragen. Im Zustand „unterbrochen“ werden die zugehörigen Daten in einen Puffer umgeleitet und können später, wenn sich der Informationsfluss wieder im Zustand „wird übertragen“ befindet, wieder aus dem Puffer geholt und übertragen werden. Ein Informationsfluss hat ähnliche Attribute wie ein DTO [KALR95].

DTOs lassen sich zu Gruppen zusammenfassen. So lässt sich z. B. eine Gruppe aller DTOs definieren, die aus ein und derselben Anwendung stammen. Statt eine Operation (z. B. das Setzen des Prioritäts-Attributes oder das Löschen) nur auf einem DTO aufzurufen, kann der Benutzer ersatzweise eine Operation auf einer Gruppe von DTOs aufrufen. Der Benutzer hat so weniger Arbeit [KALR95].

Die DTOs, die sich nicht im Zustand „wird übertragen“ befinden, werden in Warteschlangen verwaltet, die dem Benutzer auch als solche angezeigt werden können. Jede dieser Warteschlangen enthält die DTOs einer bestimmten Gruppe, die sich in einem bestimmten Zustand befinden. Beispielsweise enthält eine Warteschlange auf dem mobilen Rechner alle „E-Mails“ im Zustand „unterbrochen“. Eine andere Warteschlange auf dem MCH könnte alle Bilder eines zu übertragenden WWW-Dokument im Zustand „aufgeschoben“ enthalten.

Die Reihenfolge der DTOs in einer Warteschlange wird (unter anderem) durch die Belegung des Prioriätsattributs beeinflusst. Ist ein DTO am Kopf der Warteschlange angelangt, so wird es als nächstes der Datentransportschicht übergeben.

Der Benutzer kann nun DTOs von einer Warteschlange in eine andere verschieben, DTOs in Warteschlangen einfügen oder auch aus ihnen herauslöschen. Auch die Reihenfolge innerhalb einer Warteschlange kann er manipulieren [KALR95].

5.2.5 Herkunft und Bestimmung der Attribute

Übergibt ein Agent oder ein Proxy der Datentransferschicht ein DTO, so gibt es drei Möglichkeiten, woher die Werte eines DTO-Attributes stammen können:

- Die Attributwerte eines DTO können aus dem statischen Benutzerprofil oder – genauer – abhängig von der Anwendung, von der das DTO stammt, aus dem entsprechenden Anwendungsprofil übernommen werden. Falls sich der Sender (bzw. Empfänger) eines DTO im Internet befindet, können zudem IP-Adresse und (falls nötig) Portnummer des Senders (bzw. Empfängers) bei der Auswahl der Attributwerte eine Rolle spielen.
- Der Benutzer kann die Attributwerte eines DTO explizit setzen.
- Falls die Anwendung, von der das DTO stammt, speziell für die Mowgli-Architektur entwickelt wurde, so kennt diese Anwendung die (im Vergleich zu einem BSD-Socket) erweiterten Funktionalitäten des „Mowgli Socket Interface“ und kann die Werte der DTO-Attribute dynamisch bestimmen.

Übergibt die Datentransferschicht ein DTO an einen Kanal der Datentransportschicht, so hängen die Attributwerte dieses Kanals von den Attributwerten des DTOs ab. Die Attribute eines DTOs werden also auf die Attribute des zugehörigen Kanals abbgebildet.

5.3 Agent-Proxy-Schicht

Dieses Kapitel ist den bereits in Abschnitt 4 eingeführten Agent-Proxy-Teams und ihren Konzepten zur anwendungsspezifischen Optimierung gewidmet.

5.3.1 Arten von Agent-Proxy-Teams

Prinzipiell gibt es 3 Arten von Agent-Proxy-Teams:

- Master-Agent und Master-Proxy
- generischer Agent und generischer Proxy
- maßgeschneiderter („customized“) Agent und maßgeschneiderter („customized“) Proxy

Ruft eine Anwendung das Mowgli Socket API auf und möchte eine TCP-Verbindung aufbauen, so informiert im Regelfall der Master-Agent den Master-Proxy. Dieser erledigt dann wie bereits beschrieben als Stellvertreter für die Anwendung die eigentliche Aufgabe. Über die Funkstrecke, die nur wenig Bandbreite zur Verfügung stellen kann, versucht man hierbei und auch sonst (ähnlich wie in der Datentransportschicht) nur die allernötigsten Daten übertragen. Das zu übertragende Datenvolumen lässt sich so signifikant reduziert (vgl. weiter unten).

Ist eine TCP-Verbindung zum Kommunikationspartner im Festnetz (z. B. im Internet) erfolgreich aufgebaut worden, so wird für diese TCP-Verbindung ein generisches Agent-Proxy-Team erzeugt, das dann alle weiteren Aufgaben übernimmt, die diese Verbindung betreffen. Bereits existierende kommunizierende Anwendungen erhalten damit sozusagen einen Ersatz für die normalerweise von ihnen verwendeten Sockets.

Um UDP-Datagramme kümmert sich der Master-Agent und der Master-Proxy.

Unter einem maßgeschneiderten („customized“) Agent-Proxy-Team ist ein für das Protokoll einer Anwendung maßgeschneiderter Agent zu verstehen, der mit einem passenden Proxy zusammenarbeitet. Dieses Team weiß über die Semantik der zugehörigen Anwendung Bescheid und nutzt sie aus. Hieraus ergeben sich die folgenden Vorteile (vgl. Beispiel in Abschnitt 5.3.3) [LHKR96]:

1. Das zu übertragende Datenvolumen, insbesondere der Overhead des anwendungsspezifischen Protokolls kann auf ein Minimum reduziert werden, indem entbehrliche oder redundante Information einfach weggelassen wird.
2. Daten können typspezifisch komprimiert werden.
3. Falls die GSM-Verbindung zum Festnetz (aufgrund schlechter Bedingungen) vorübergehend unterbrochen wird, kann die Agent-Proxy-Schicht anwendungsspezifisch „wiederaufsetzen“ (recovery) [AKLR96].
4. Falls (vorübergehend) keine GSM-Verbindung besteht, können Agent und Proxy in einem getrennten bzw. in einem fast getrennten Modus selbstständig weiterarbeiten (vgl. Abschnitt 5.3.3).

5. Kommunizierende Anwendungen übertragen heutzutage oft größere Datenmengen, die in verkabelten Netzen kaum bzw. (noch) nicht zu Engpässen führen. Werden solch umfangreiche Datenmengen über Funkstrecke übertragen, so werden wegen der mässigen technischen Leistungsfähigkeit der GSM-Verbindung Antwortzeiten oft quälend lang. Deshalb und auch, um die teure GSM-Verbindung ökonomisch zu nutzen, benötigt der Benutzer zum Umgang mit den knappen Ressourcen eine feinkörnige Kontrolle der Datenübertragung.

   Bei einem E-Mail-spezifischen Agent-Proxy-Team kann der Benutzer z. B. bestimmten, welche Mails in welcher Reihenfolge vom Mail-Server auf den Klienten des mobilen Rechners heruntergeladen werden. Kleine bzw. besonders wichtige Mails mit bestimmten Eigenschaften (z. B. von einem gewissen Absender) können beispielsweise zuerst heruntergeladen werden.

Der maßgeschneiderte Agent kann in die zugehörige (Klienten-)Anwendung integriert werden. Da der entsprechende Proxy dann fest zur Anwendung gehört, besteht diese als Konsequenz quasi aus zwei Teilen.

Von Nachteil ist dabei wohl, dass Anwendung, Mowgli-Socket-API und Agent-Proxy-Schicht monolithisch zu einer Einheit verschmelzen, so dass dabei die Grenzen zwischen den Einzelkomponenten verwischt werden. Es kann dann nicht mehr von einem Socket im herkömmlichen Sinne gesprochen werden.

5.3.2 Der (fast) getrennter Modus

Normalerweise, d. h. unter guten Sende- bzw. Empfangsbedingungen, arbeiten Agent und Proxy in einem „stark verbundenen Modus“ miteinander.

Agent und Proxy sind in der Lage, zeitweilig in einem „getrennten Modus“, d. h. ohne bestehende GSM-Verbindung, auf beiden Seiten der Funkstrecke selbstständig weiter zu arbeiten. Die Anwendungen auf dem mobilen Rechner bzw. die Kommunikationspartner im Festnetz merken davon im Idealfall nichts und können mit den Stellvertretern (Agent und Proxy) zumindest eine Zeitlang weiterarbeiten, ohne dass eine GSM-Verbindung zu den „Stellzuvertretenden“ besteht [AKLR96].

Die Entscheidung, wann sich das System im „getrennten Modus“ befindet, hängt erstens von einigen Datenkanalattributen (z. B. „GSM-Verbindungserlaubnis“) und zweitens von den Bedingungsattributen der in Auftrag gegebenen DTOs ab. Dadurch spielen bei der Entscheidung die momentanen Sende- bzw. Empfangsbedingungen implizit eine große Rolle (vgl. Abschnitte 5.1.2 und 5.2.3).

Alternativ zum getrennten Modus kann das Agent-Proxy-Team auch in einem „fast getrennten Modus“ arbeiten. Die GSM-Verbindung wird nur dann für kurze Zeit aufgebaut, wenn sie wirklich gebraucht wird bzw. dringend erforderlich ist. Agent und Proxy kommunizieren miteinander nach transparenten, festen, vorher vereinbarten Regeln.

Als Szenario könnte man sich z. B. einen intelligenten Proxy vorstellen, der selbstständig einige vom Agenten erhaltene Aufgaben erfüllt und verschiedene Datenpakete aus dem Internet besorgt. Auch wenn der Proxy schon einige Aufgaben erledigt hat, wird der Agent allerdings erst dann benachrichtigt, wenn der Proxy etwas besonders „Interessantes“ vorzuweisen hat, das bestimmten Filterbedingungen genügt. Eine solche Benachrichtigung, die evtl. genaue Informationen über vom Proxy „verbuchte Erfolge“ enthält, könnte auch per SMS stattfinden [AKLR96]. (SMS steht hier für „short message service“, einen populären GSM-Dienst.)

5.3.3 Ein WWW-spezifisches Agent-Proxy-Team als Beispiel

Dieser Abschnitt beschreibt, wie ein (für das WWW) maßgeschneidertes Agent-Proxy-Team die ihm bekannte Anwendungssemantik ausnutzt. Das Agent-Proxy-Team wurde im Rahmen des Mowgli-Projektes tatsächlich implementiert. Dabei wurde der Agent in einen WWW-Klienten integriert, der die „neuen" Funktionalitäten der Mowgli-Architektur kennt und nutzt.

- In WWW-Dokumente eingebettete Bilder werden typspezifisch komprimiert, indem Auflösung und Farbtiefe vor der Übertragung über die Funkstrecke reduzuziert werden (vgl. Punkt 2 in Abschnitt 5.3.1).
- Möchte der Benutzer ein WWW-Dokument vom Festnetz herunterladen, das mehrere, verschieden große Bilder beinhaltet, so kann das WWW-spezifische Agent-Proxy-Team so eingestellt werden, dass der Text und kleine Bilder sofort geladen werden. Während der Benutzer sich einen Eindruck von dem Dokument verschaffen kann, werden die mittelgroßen Bilder im Hintergrund geladen. Die großen Bilder werden nur dann geladen, wenn der Benutzer es ausdrücklich wünscht. Die tatsächlich über die Funkstrecke übertragenenen Bilder werden so über ihre Größe aus allen eingebetteten Bildern des WWW-Dokuments intelligent herausgefiltert [LHKR96] (vgl. Punkt 5 in Abschnitt 5.3.1).
- Zwischen Agent und Proxy – also auf der Funkstrecke – wird das HTTP (hypertext transfer protocol) durch ein gleichwertiges, alternatives Protokoll namens MHTTP (Mowgli HTTP) ersetzt. MHTTP bietet folgende Vorteile (vgl. Punkt 1 in Abschnitt 5.3.1):
  - MTTHP ist im Gegensatz zu HTTP nicht lesbar, sondern binär codiert. Die Größe der Protokollheader lässt sich so auf ungefähr ein Fünftel reduzieren. Die schmale Bandbreite der GSM-Verbindung wird dadurch weniger in Anspruch genommen.
  - Anders als bei HTTP wird bei der Verwendung von MHTTP nicht für jedes Objekt, das in ein zu übertragendes WWW-Dokument eingebettet ist, eine eigene TCP-Verbindung über die Funkstrecke hinweg auf und wieder abgebaut. Denn um die notwendigen TCP-Verbindungen zum Webserver kann sich der Proxy kümmern. Die kurzlebigen TCP-Verbindungen beschränken sich dann auf das Festnetz. Die Verbindung zwischen WWW-Agent und WWW-Proxy zur Übertragung des WWW-Dokuments ist langlebiger. Der Einsatz von MHTTP verkürzt so wegen der hohen Verzögerung der GSM-Verbindung die Antwortzeiten beim „Download" von WWW-Dokumenten [LAKL+95].

    In ein Webdokument eingebettete Bilder lädt der Proxy beispielsweise schon vom Webserver, während parallel noch der Text mit den Verweisen auf die Bilder über die Funkstrecke zum WWW-Agenten übertragen wird. Ist die Übertragung des Textes zum Agenten abgeschlossen, kann sogleich mit der Bildübertragung fortgefahren werden, da die Bilder dann schon beim Proxy in einem Puffer vorliegen. Die schmale Bandbreite der GSM-Verbindung wird so ohne Unterbrechung genutzt, ohne dass weitere HTTP-Anfragen erfolgen müssen [LAKL+95].
  - Es wird darauf verzichtet, bei jedem Zugriff auf ein WWW-Dokument, dessen Gültigkeit zu überprüfen. Zum einen ist dies im „getrennten Modus" nicht möglich. Zum anderen würde dies wegen der hohen Latenz der GSM-Verbindung die Antwortzeit bei der Übertragung eines Webdokumentes spürbar verlängern [LHKR96].
- Besondere Cache-Funktionalitäten und das Laden der als nächstes benötigten Dokumente im Voraus („prefetching") ermöglichen dem Benutzer ein komfortables Arbeiten ohne bestehende GSM-Verbindung (vgl. Punkt 4 in Abschnitt 5.3.1):

- Während der Benutzer noch mit dem Lesen eines WWW-Dokumentes beschäftigt ist, hat er die Möglichkeit, die Verweise auf die als nächstes benötigten WWW-Dokumente zu markieren. Diese Dokumente werden dann im Hintergrund in den Cache des WWW-spezifischen Agenten übertragen. Möchte sich der Benutzer später den vorher bereits markierten Dokumenten zuwenden, so ist die Antwortzeit in der Regel erheblich kürzer. Und falls die GSM-Verbindung aufgrund schlechter Empfangsbedingungen zwischenzeitlich abgebrochen sein sollte und die Übertragung des markierten Dokuments erfolgreich abgeschlossen wurde, kann der Benutzer im „getrennten Modus“ weiterarbeiten [LAKL+95].
- WWW-Dokumente, die sich im persistenten Cache befinden, werden nicht automatisch gelöscht. Der Benutzer muss nicht mehr benötigte Dokumente explizit freigeben. Alternativ hierzu kann besteht die Möglichkeit, dass WWW-Dokumente im Normalfall automatisch aus dem Cache gelöscht werden. Jedoch kann der Benutzer dann Dokumente durch eine Sperre vor dem Löschen schützen. Eine solche Sperre kann selbstverständlich wieder aufgehoben werden [LHKR96].
- In einem WWW-Dokument werden Verweise auf andere WWW-Dokumente hervorgehoben angezeigt, wenn sich die Dokumente, auf die verwiesen wird, noch im Cache bzw. durch Übertragung im Voraus bereits im Cache befinden. Der Benutzer kann sich so – besonders im „getrennten Modus“ – sicher innerhalb der „Grenzen“ des Caches bewegen [LHKR96].

5.4 Mowgli Socket Interface

Die mit dem Festnetz kommunizierenden Anwendungen des mobilen Rechners können sämtliche in den Abschnitten 5.1 bis 5.3 beschriebendenen Telekommunikationsdienste der Mowgli-Architektur direkt über ein API, das „Mowgli Socket Interface“ (MSI), aufrufen [KALR95]. Dies ist in Abbildung 5 grafisch veranschaulicht.

Wie bereits erwähnt ist dabei die Verwendung des Dienstes der Datentransferschicht optional, und die Funktionalität der Agent-Proxy-Schicht kann bereits in eine kommunizierende Anwendung integriert sein.

Das API ist eine Erweiterung des verbreiteten „BSD Socket Interface“. Die Funktionalität eines BSD-Sockets wird von der Mowgli-Architektur natürlich ebenso zur Verfügung gestellt. Bereits auf dem mobilen Rechner befindliche Anwendungen, die normalerweise z. B. auf BSD-Sockets aufbauen, können daher auch das erweiterte API ohne Modifikation verwenden. Damit ist eine weitere Anforderung aus Abschnitt 1.3 erfüllt [AKLR96].

Das MSI stellt aber noch weitere Funktionalitäten zur Verfügung, die von einem konventionellen TCP/IP-Socket (wie dem BSD Socket) nicht unterstützt werden.

Das MSI hat dafür eine Reihe von Attributen, durch die das Verhalten der Kommunikationsdienste gesteuert bzw. beeinflusst werden kann. Aus den Werten dieser Attribute ergeben sich unter anderem die Attributwerte der DTO aus Abschnitt 5.2.3 bzw. der Datenkanäle aus Abschnitt 5.1.2. Den Datenpaketen, die durch das MSI gereicht werden, können so beispielsweise unterschiedliche Prioritäten zugewiesen werden. Zusätzlich wird z. B. abhängig von der Belegung der Socketattribute ein zu übertragendes Datenpaket an ein bestimmtes Agent-Proxy-Team übergeben.

Falls der Benutzer eine bereits existierende kommunizierende Anwendung verwenden möchte, jedoch keinen Quellcode ändern kann oder möchte, so kann er die zur Anwendung gehörigen Socketattribute extern und statisch konfigurieren. Neuere Anwendungen, die speziell für mobile Rechner entwickelt wurden, und die das MSI kennen und ausnutzen, können hingegen die Socketattribute direkt und dynamisch setzen (vgl. Abschnitt 5.2.5).

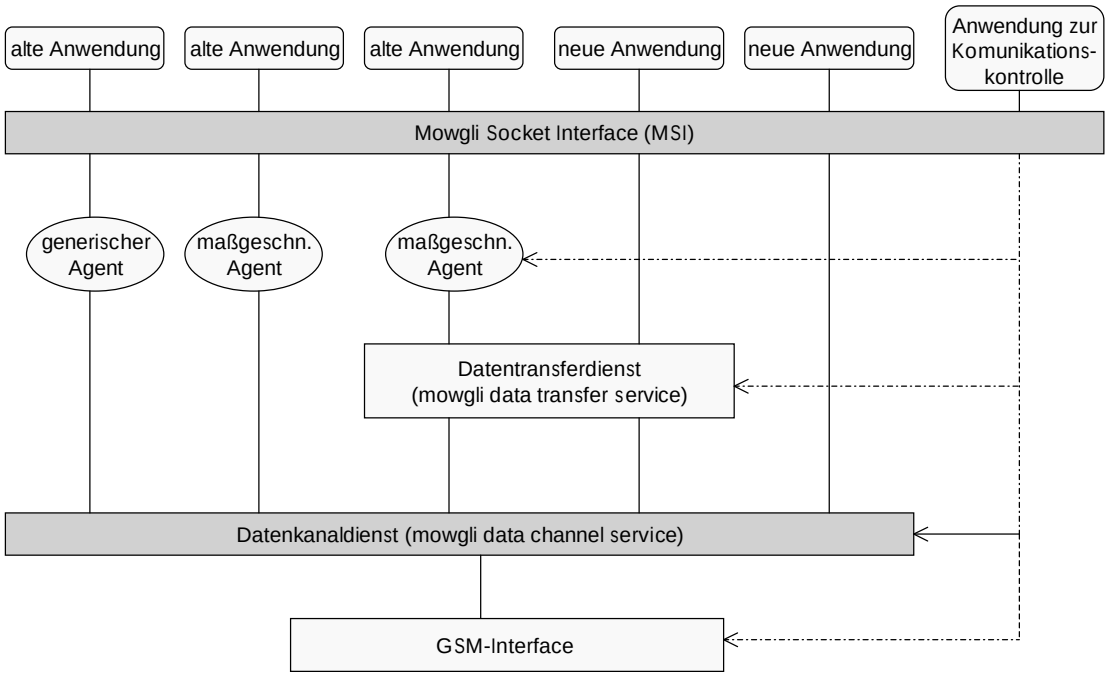


Abbildung 5: Zugriff auf die Dienste über das Mowgli Socket Interface

Neben Operationen, die die reguläre Übertragung von Daten betreffen, gibt es noch Kontrolloperationen, mit deren Hilfe sich die GSM-Verbindung sowie die Vorgänge in sämtlichen Schichten direkt manipulieren lassen. Sie haben höchste Priorität und werden sofort ausgeführt. Außerdem lassen sie sich nicht unter- bzw. abbrechen, sondern sind atomar. Diese Kontrolloperationen können außer von „neuen“ kommunizierenden Anwendungen auch von einer Kommunikationskontrollanwendung aufgerufen werden (vgl. Abschnitt 5.5).

5.5 Benutzer-Schnittstelle

Bevor der Benutzer den Auftrag absetzt, eine Datenübertragung durchzuführen, kann er sich über die voraussichtliche Übertragungsdauer unter den momentanen Bedingungen informieren. Der Benutzer kann sich dann ad hoc entscheiden, ob und wann er die Übertragung durchführen möchte.

Dies ist dann abhängig

- von der bereits erwähnten Übertragungsdauer unter den momentanen Bedingungen,
- von der Wichtigkeit der evtl. zu übertragenden Daten und
- vom momentanen Telefontarif.

Der Benutzer erhält so also die Möglichkeit gewisse, evtl. teure Übertragungsvorgänge direkt zu steuern [AKLR96].

Alternativ dazu kann der Benutzer auch die Mowgli-Architektur bemächtigen, diese Entscheidung unter vorher festgelegten Kriterien selbstständig durchzuführen. Dazu kann der Benutzer der Mowgli-Architektur eine Richtlinie („policy“) vorgeben, aus der das System ableiten kann, wie es automatisch verfahren soll. Die Richtlinie entspricht der Konfiguration der MSI-Attribute in Abschnitt 5.4. In ihr kann z. B. auch das Systemverhalten festgelegt werden, falls die GSM-Verbindung zum Festznetz abbricht.

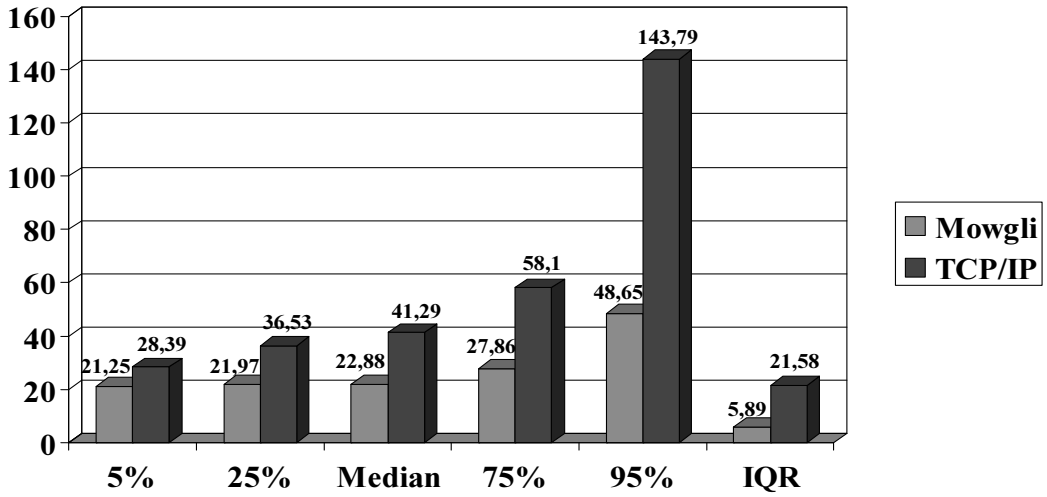


Abbildung 6: vergleichende, statistische Auswertung der gemessenen Antwortzeiten

Falls der Benutzer oder die Datentransferschicht entscheiden, eine Datenübertragung nicht oder erst zu einem späteren Zeitpunkt durchzuführen, z. B. falls eine (akzeptable) Anbindung des portablen Rechners an das Festnetz zeitweilig nicht möglich ist, kann der Benutzer evtl. in einem „getrennten Modus“ weiterarbeiten, so dass bis zu einem gewissen Grade nicht auffällt, dass momentan keine GSM-Verbindung zum Festnetz aufgebaut ist (vgl. Abschnitt 5.3.2).

Insbesondere für den Fall, dass der Benutzer ausschliesslich bereits existierende kommunizierende Anwendungen benutzt, stellt eine „Anwendung zur Kommunikationskontrolle“ dem Benutzer eine (Benutzer-)Schnittstelle zur Verfügung, mit deren Hilfe er die Kontrollfunktionen des MSI (vgl. Abschnitt 5.4) nutzen und z. B. die GSM-Verbindung oder auch gewisse Attribute von DTOs bzw. Datenkanälen direkt manipulieren kann. Außerdem lässt sich mit dieser Anwendung auch die Konfiguration der Socketattribute einstellen.

6 Experimentelle Messungen

Im Vergleich zu konventionellen Kommunikationsarchitekturen, bei denen in der Regel TCP und IP verwendet werden, haben sich die technischen Leistungen unter Einsatz der Mowgli-Architektur deutlich erhöht. Messungen im Rahmen des Mowgli-Projektes haben dies ergeben.

Die Messungen haben weiterhin gezeigt, dass vor allem auch die Anpassung eines Agent-Proxy-Teams an eine bestimmte Anwendung die technischen Leistungen in einem bemerkenswerten Maße erhöht. Ein vom Mowgli-Projekt implementiertes Agent-Proxy-Team, das speziell für das Browsen im WWW entwickelt wurde (vgl. Abschnitt 5.3.3), hat diese Messungen ermöglicht. Wie bereits erwähnt sehen andere Architekturen, die ähnliche Ziele wie die Mowgli-Architektur verfolgen, keine anwendungsspezifischen Optimierungen vor, wie sie in einem maßgeschneiderten Agent-Proxy-Teams realisiert werden (vgl. Abschnitt 5).

Exemplarisch werden nun im Folgenden zwei Versuchsreihen aus dem Mowgli-Projekt vorgestellt, in denen jeweils 100 Mal ein WWW-Dokument (79.920 Bytes) mit sieben eingebundenen Bildern (4x 384 Bytes, 2x 2.048 Bytes und 1x 10.240 Bytes) von einem WWW-Server in Toronto auf einen mobilen Rechner geladen wurde. Der MCH („mowgli connection host“) befand sich dabei in einem LAN der Universität Helsinki [LHKR96].

Bei der ersten Versuchsreihe kamen auf der Funkstrecke die Mowgli-Architektur und das MHTTP zum Einsatz, während bei der zweiten Versuchsreihe dort nur eine konventionelle Architektur, basierend auf HTTP sowie TCP/IP, verwendet wurde. Selbstverständlich war

man bemüht, bei beiden Versuchsreihen gute und vergleichbare Übertragungsbedingungen zu schaffen.

Das Balkendiagramm in Abbildung 6 zeigt eine vergleichende, statistische Auswertung der gemessenen Antwortzeiten. Außer dem Median wurde bei beiden Versuchsreihen auch dessen Verallgemeinerung, das α-Quantil [Hind93], für $\alpha = 5\ \%$, $25\ \%$, $75\ \%$ und $95\ \%$ ermittelt. Falls α die Werte $25\ \%$ bzw. $75\ \%$ annimmt, spricht man auch vom unteren bzw. oberen Quartil. Die Differenz dieser beiden Werte bezeichnet man als den Quartilsabstand („inter quartile range“, IQR) Ähnlich, wie der Median dem arithmetischen Mittelwert entspricht, ist der Quartilsabstand mit der Standardabweichung vergleichbar. Median und Quartilsabstand haben jedoch den Vorteil, dass sie unempfindlicher gegen Ausreißer sind.

Ein Blick auf das Balkendiagramm in Abbildung 6 zeigt, dass die Antwortzeiten bei der zweiten Versuchsreihe in der Regel ungefähr 1.5 bis 2 Mal so lang sind wie bei der ersten Versuchsreihe, in der MHTTP und die Mowgli-Architektur verwendet wurden. Der Quartilsabstand ist bei der ersten Versuchsreihe außerdem wesentlich kleiner als bei der zweiten, was auf die höhere Fehlertoleranz bei der Mowgli-Architektur zurückgeführt werden kann. Denn Störungen schlagen bei der Mowgli-Architektur weniger zu Buche [LHKR96].

7 Kritik und Ausblick

Um die Mowgli-Architektur für möglichst viele Benutzer bzw. für einen Benutzer an möglichst vielen Orten nutzbar zu machen, müssen die folgenden beiden Voraussetzungen erfüllt sein:

- Es müssen möglichst viele Einwahlknoten mit der Mowgli-Architektur ausgestattet werden, so dass diese dann als MCH eingesetzt werden können. Die „Mowgli-Einwahlknoten“ sollten dabei möglichst so über die Fläche verteilt sein, dass einem Benutzer stets ein möglichst naher Einwahlknoten zur Verfügung gestellt werden kann.
- Die Mowgli-Architektur passt sich zwar bereits existierenden kommunizierenden Anwendungen, die z. B. auf BSD-Sockets aufbauen, an. Um die neuartigen Funktionalitäten der Architektur optimal auszunutzen zu können, muss jedoch neue Kommunikationssoftware entwickelt bzw. bestehende weiterentwickelt werden (vgl. Abschnitt 5.4).

Diese beiden Voraussetzungen sind jedoch wechselseitig abhängig voneinander. Die Ausstattung vieler Einwahlknoten lohnt sich erst, wenn es genügend Kommunikationssoftware gibt, um diese gut auszunutzen. Umgekehrt verhält sieht es sich ähnlich: Die Kosten, die durch die oben erwähnte (Weiter-)Entwicklung von Kommunikationssoftware entstehen würden, lassen sich erst dann rechtfertigen, wenn es genügend Einwahlknoten gibt, bei denen diese Software eingesetzt werden kann. Es handelt sich hier daher sozusagen um eine Art „Huhn-Ei-Problem“.

In den nächsten Jahren sollen nun einige neue Technologien zum Einsatz kommen, die mehr als 9.600 Bit/s Bandbreite zur Verfügung stellen. Ob der Einsatz von GSM-Systemen zur drahtlosen Übertragung von Daten dann in Zukunft noch sinnvoll ist, wird sich zeigen. Es folgt nun eine Aufzählung der Technologien:

- Noch 1999 soll die Spezifikation von GSM Phase II umgesetzt werden, das bei reduzierter Fehlerkorrektur 14.400 Bit/s anbietet. Durch eine besondere Form der Datenkompression lässt sich hierbei der Durchsatz bei gut komprimierbaren Daten sogar verdoppeln [Koss99].

- HSCSD (high-speed circuit switched data service) kann zwei bis acht Datenkanäle von 9.600 Bit/s oder 14.400 Bit/s bündeln und soll ebenfalls bis Ende 1999 einsatzbereit sein [Koss99].
- T-Mobil will bis Mitte des Jahres 2.000 GPRS (general packet radio service) implementieren, das Funkkapazitäten besser nutzt und so zunächst 50 kBit/s, später 100 kBit/s liefern soll [Koss99].
- Im Jahr 2.001 soll dann UMTS (universal mobile telecommunication system) mit 2 Mbit/s zum Einsatz bereit stehen. UMTS soll als weltumspannendes Mobilfunknetz die Nachfolge von GSM antreten [Rink99].

Vergleicht man nun jedoch die Bandbreiten der aufgezählten Technologien mit den Bandbreiten, die man in naher Zukunft von fest verdrahteten Netzwerken erwarten kann, so wächst die Diskrepanz zwischen der technischen Leistungsfähigkeit bei der drahtlosen Datenübertragung und der technischen Leistungsfähigkeit in Festznetzen sogar noch. Zumindest die in dieser Seminararbeit vorgestellten Konzepte dürften deshalb auch in Zukunft noch genügend Verwendung finden.

Literatur

[AKLR96] Timo Alanko, Markku Kojo, Heimo Laamanen und Kimmo Raatikainen. Mobile computing based on GSM: The Mowgli approach. In J.E. Encarnacao und J.M. Rabaey (Hrsg.), *Mobile Communications - Technology, Tools, Applications, Authentication and Security*, S. 151 – 158. Chapman & Hall, 1996.

[Hind93] Dr. Dr. K. Hinderer. *Stochastik für Informatiker und Ingenieure.* Universität Karlsruhe (Institut für Mathematische Stochastik). Juli 1993.

[KALR95] Markku Kojo, Timo Alanko, Mika Liljeberg und Kimmo Raatikainen. *Enhanced Communication Services for Mobile TCP/IP Networking*, Band No. C-1995-15 der *Series of Publications C.* University of Helsinki (Department of Computer Science). April 1995.

[KKLR96] Jani Kiiskinen, Markku Kojo, Mika Liljeberg und Kimmo Raatikainen. *Data Channel Service for Wireless Telephone Links*, Band No. C-1996-1 der *Series of Publications C.* University of Helsinki (Department of Computer Science). Januar 1996.

[Koss99] Axel Kossel. Internet im Gepäck. *c't* Band 11, Mai 1999, S. 114 – 116.

[LAKL+95] Mika Liljeberg, Timo Alanko, Markku Kojo, Heimo Laamanen und Kimmo Raatikainen. Optimizing World-Wide Web for Weakly Connected Mobile Workstations: An Indirect Approach. In *Proceedings of the 2nd International Workshop on Services in Distrbiuted and Networked Environments (SDNE '95)*, Juni 1995.

[LHKR96] Mika Liljeberg, Heikki Helin, Markku Kojo und Kimmo Raatikainen. *Enhanced Services for World-Wide Web in Mobile WAN Environment*, Band No. C-1996-28 der *Series of Publications C.* University of Helsinki (Department of Computer Science). April 1996.

[Rink99] Dr. Jürgen Rink. Nirgendwo mehr allein. *c't* Band 11, Mai 1999, S. 92 – 95.

[Tane98] Andrew S. Tanenbaum. *Computernetzwerke.* Prentice Hall. 3. Auflage, 1998.

Mole - ein System für mobile Agenten

Fabian Just

Kurzfassung

Der nachfolgende Seminarbericht befaßt sich mit dem Konzept von 'mobilen Agenten'-Systemen. Nach einer allgemeinen Einführung in die Arbeitsweise eines Systems für mobile Agenten werden anhand von Mole exemplarisch die Merkmale eines solchen Systems erläutert. Mole ist ein Java-basiertes System, das von Straßer, Baumann und Hohl an der Universität Stuttgart entwickelt wurde. Nachdem auf die einzelnen Aspekte eingegangen wurde, wendet sich der Seminarbericht den Schwächen und Stärken solcher Systeme zu und gibt einen kurzen Ausblick auf zukünftige Entwicklungen.

1 Einleitung

'Mobile Agenten liegen im Trend' - immer wieder hört man diese Aussage und bekommt als Beispiel meistens Suchroboter präsentiert. Doch wie funktionieren mobile Agenten eigentlich?

Mobile Agenten sind Prozesse, die im Auftrag ihres Benutzers auf verteilten Systemen arbeiten. Dazu ist es nötig, daß sich diese Prozesse frei von einem Knoten des verteilten Systems zu einem anderen bewegen können. Ist ein mobiler Agent erst einmal 'unterwegs', ist es nicht mehr nötig, daß der Benutzer ständig mit dem verteilten System verbunden ist. Diese Tatsache hat, wie man später sehen wird, enorme Vorteile.

Um die oben aufgeführten Funktionen zu ermöglichen, muß jeder Knoten in dem verteilten System eine verläßliche Infrastuktur bzw. Plattform bieten. Diese Plattform besteht meist aus einer virtuellen Maschine (z.B. *java virtual machine*) und ist in der Lage, zwei Arten von Objekten zu verwalten: Ablaufumgebungen (in Mole 'Plätze' genannt) und Agenten. Die Ablaufumgebungen bieten den Agenten eine Infrastruktur, damit diese ausgeführt werden können. Objekte vom Typ 'Agent' können sich in solchen Ablaufumgebungen aufhalten, von einer Ablaufumgebung zu einer anderen wandern und können untereinander kommunizieren. Die Agenten lassen sich weiter in zwei verschiedene Gattungen aufsplitten: mobile Agenten und stationäre Agenten (auch Serviceagenten genannt). Während die mobilen Agenten, wie oben beschrieben, zwischen Netzwerkknoten wandern können, sind stationäre Agenten an ihre Ablaufumgebung gebunden. Sie bieten ihren mobilen 'Kollegen' verschiedene Dienste und Systemressourcen an.

In dem nachfolgenden Kapitel soll nun Mole vorgestellt werden. Mole ist ein System für mobile Agenten, das auf JAVA basiert. Neben Mole (1996), das an der Universität Stuttgart von Straßer, Baumann und Hohl entwickelt wurde, existiert z.B. Aglet vom IBM Aglet Workbench Team (1997), das ebenfalls auf JAVA basiert. Darüber hinaus gibt es natürlich auch Systeme wie Agent Tcl von Gray (1995,1996), die auf anderen Programmiersprachen, wie z.B. Tcl, aufbauen.

2 Wie funktioniert Mole ?

2.1 Überblick

Mole benutzt das Konzept der Plätze (entspricht den Ablaufumgebungen) und Agenten. Ein Agentensystem besteht aus mehreren Plätzen, die verschiedene Dienste in Form von Service-Agenten anbieten. Die Agenten können zwischen diesen Plätzen wechseln. Sie können Service-Agenten in Anspruch nehmen und mit anderen Agenten kommunizieren. Diese Kommunikation ist nicht lokal auf einen Platz beschränkt, sondern kann auch zwischen Agenten stattfinden, die sich in verschiedenen Plätzen befinden. Dazu ist es natürlich notwendig, daß Agenten eine global eindeutige Bezeichnung erhalten. Diese Bezeichnung wird dem Agenten während seiner Entstehung vom Platz seiner Entstehung zugeordnet. Dieser Identifier ändert sich während der Lebenszeit des Agenten nicht mehr, selbst wenn der Agent mehrmals zwischen Plätzen wandert. Außerdem ist es möglich, aus dem Bezeichner des Agenten seinen Entstehungsort abzuleiten. Im Abschnitt 2.4 wird auf die Vorteile dieses Verfahrens eingegangen.

Ein einzelner Netzwerkknoten kann mehrere Plätze anbieten. Dies macht z.B. Sinn, wenn auf einem Server die Angebote von verschiedenen Firmen dargeboten werden sollen. Jede Firma erhält dann einen eigenen Platz, auf dem sie eigene Service-Agenten einrichten kann. Ein Platz kann sich jedoch nicht über mehere Netzwerkknoten erstrecken. Ein Platz wird als 'connected' bezeichnet, wenn das darunterliegende System permanent mit dem Netz verbunden ist. Bei nur zeitweiliger Anbindung wird ein Platz als 'associated' bezeichnet.

Die folgende Abbildung 1 zeigt die wichtigsten Komponenten von Mole: die Plätze (engl. **place**), die mobilen Agenten und die Service-Agenten. Diese Service-Agenten bieten den mobilen Agenten Dienste an. Sie stellen für die mobilen Agenten die einzige Schnittstelle zum darunterliegen System dar. Wie in der Abbildung gezeigt, ist z.B. der Zugriff auf die Datenbank des Servers nur über den zugeordneten Service-Agenten möglich.

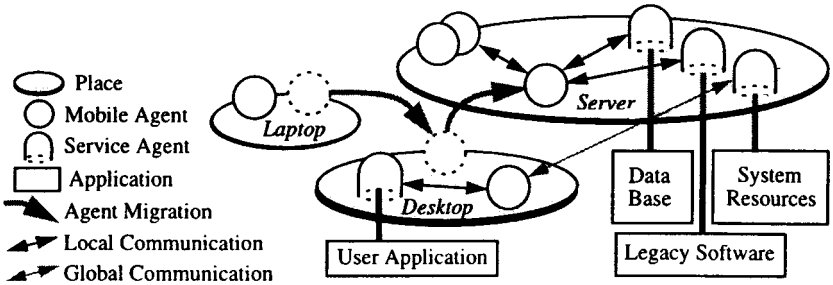


Abbildung 1: das Mole-System im Überblick [BHRS+98a]

2.2 Lebenszyklus eines Agenten

Nachdem ein Agent kreiert wurde, wird seine `init()`-Methode aufgerufen. Die Parameter beim Aufruf des Agenten werden dieser Methode übergeben. Er befindet sich nun im selben Zustand wie ein Agent, der gerade zu diesem Platz gewandert ist. Im nächsten Schritt wird der Agent in das System eingeführt. Er wird den anderen Agenten des Platzes bekannt gemacht, diese dürfen jedoch noch nicht mit ihm kommunizieren. Nun wird die `prepare()`-Methode des Agenten aufgerufen, die es ihm ermöglicht, seine für diesen Platz spezifischen Einstellungen zu treffen. Jetzt fehlt nur noch der Aufruf der `start()`-Methode des Agenten und der Agent kann normal arbeiten (eigene Threads starten etc.).

Der Agent kann nun jederzeit `migrateTo()` aufrufen, um zu einem anderen Platz zu migrieren. Sobald an dem neuen Platz `prepare()` und `start()` erfolgreich ausgeführt wurden, wird der

Agent vom bisherigen Platz entfernt. Anderfalls erhählt er eine Fehlermeldung und wird nach der `migrateTo()`-Stelle fortgesetzt.

Hat der Agent das Ende seines Lebenszykluses errreicht, kann er die `die()`-Methode aufrufen. Seine threads werden dann gestoppt, er wird vom Platz entfernt und gelöscht.

2.3 Wanderung von Agenten

Bei der Wanderung von Agenten gibt es mehrere Konzepte, die unterschiedliche Arten der Wanderungen durchführen. Dies liegt daran, daß für die Wanderung eines Agenten zu einem anderen Platz nicht nur der Code des Agenten übertragen werden muß, sondern auch sein Zustand. Der Zustand laßt sich dabei weiter aufschlüsseln in 'Zustand der Daten' und 'Ausführungszustand'.

Im folgenden werden drei Konzepte vorgestellt, deren Realisierung in der Reihenfolge der Aufzählung immer schwieriger wird.

Bei der 'ferngesteuerten Ausführung' (engl. **Remote Execution**) wird das Agentenprogramm auf einen anderen Platz transferiert, ohne vorher ausgeführt worden zu sein. An diesem Platz wird es ausgeführt bis es terminiert. Somit wird der Agent nur ein einziges Mal transferiert. Es ist jedoch möglich, daß der transferierte Agent während seiner Ausführung einen weiteren Agenten aufruft. Neben der 'ferngesteuerten Ausführung' gibt es eine Abwandlung mit der Bezeichnung '**Code on Demand**'. Bei der 'Remote Execution' bestimmt der fernsteuernde Teil den Zielort des Agenten. Dies ist bei der 'Code on Demand'-Variante anders. Das Ziel selbst veranlaßt den Transfer des Agenten. Diese Methode kann z.B. in Client/Server-Umgebungen für Softwareupdates eingesetzt werden. Der Client veranlaßt nach eigenem Ermessen die Übertragung eines Agenten, der neue Software enthält und auch gleich installiert.

Die 'schwache Wanderung' (engl. **Weak Migration**) ist in der Lage, bereits laufende Agenten beliebig oft auf andere Plätze zu übertragen. Dabei wird jedoch, neben dem eigentlichen Programmcode, nur der Zustand der Daten übertragen. Somit ist der Programmierer dafür verantwortlich, daß die für den Verlaufszustand relevanten Daten in Programmvariablen abgelegt werden. Außerdem muß der Programmierer eine modifizierte `start()`-Methode entwerfen, die aufgrund der decodierten Zustandsinformationen (gewonnen aus den Programmvariablen) entscheidet, an welcher Programmstelle die Ausführung nach einer Wanderung fortgesetzt werden soll. Durch dieses Verfahren reduziert sich die Anzahl der Zustände, in denen ein Agent migrieren kann.

Bei der 'starke Wanderung' (engl. **Strong Migration**) wird neben dem Agenten-Code sowohl der Zustand der Daten als auch der Verlaufszustand transferiert. Die starke Migration umfaßt somit den ganzen Agenten. Für den Programmierer ist dies natürlich sehr angenehm, da die komplette Wanderung vom darunterliegenden System transparent durchgeführt wird. Andererseits erfordert diese Art der Wanderung ein globales Modell, das den Agentenzustand beschreibt. Nur so kann an dem Zielplatz der Agent in seinen zuletzt eingenommenen Zustand versetzt werden. Außerdem benötigt man eine Transfersyntax, um die Agentenzustände zu übertragen.

Mole benutzt für seine Agenten die 'schwache Wanderung'. Dies liegt daran, daß eine normale JAVA Virtual Machine das Abfragen des Ausführungszustandes eines Threads nicht erlaubt. Da Mole aber auf jeder normalen JAVA Virtual Machine laufen soll, scheidet die 'starke Wanderung' aus. Zusätzlich ist die 'starke Wanderung'oft mit einer großen zu übertragenden Datenmenge verbunden, da der Ausführungszustand unter Umständen ein recht umfangreiches Datenpaket ist. Aus diesen Gründen kommt in Mole die 'schwache Wanderung' zum Einsatz.

2.4 Agenten ID

Wie bereits im einleitenden Teil 2.1 erwähnt, erhält jeder Agent bei seiner Entstehung eine eindeutige Bezeichnung (engl. **ID**) zugeordnet. Diese ID muß sowohl lokal als auch global eindeutig sein, damit die Kommunikation zwischen Agenten reibungslos verläuft. Aber auch für die Terminierung von Agenten ist eine eindeutige ID unerläßlich. In Mole setzt sich deshalb die Bezeichnung eines Agenten wie in der nachfolgenden Tabelle 1 dargestellt zusammen.

Komponenten einer Agenten-ID	
Anzahl Bytes	Bedeutung
4	Dynamischer Zähler; wird bei jeder neuen Agenten-ID erhöht
4	Crash-Zähler; er wird bei jedem Neustart des Systems erhöht; ein Überlauf des dynamischen Zählers führt ebenfalls zu einer Erhöhung
12	IPv6-Adresse des Systems, auf dem die Agentenplatform läuft
2	Die Port-Nummer des Platzes
2	Reserviert für zukünftigen Gebrauch (auf 0 gesetzt)

Tabelle 1: Die Tabelle gibt an, wie sich die 24 Byte lange Agenten-ID zusammensetzt

Die IPv6 - Adresse des zugrundeliegenden System erlaubt in Zusammenhang mit der Port-Nummer eine eindeutige Identifikation des Entstehungsplatzes. Die Möglichkeit aus der Agenten-ID den Ursprungsplatz abzuleiten, bietet einige Vorteile. So kann ein Platz z.B. gezielt Agenten ablehnen, die von unsicheren Plätzen kommen. Dies ist aus Sicherheitsaspekten eine nützliche Funktion. Auch z.B. bei GSM ist es nützlich, aus der ID eines Kunden einen bestimmten Platz ableiten zu können, an dem Informationen über den Aufentshaltsort des Kunden abgelegt sind.

Wie der Tabelle 1 entnommen werden kann, wird zur eindeutigen Identifikation der Agenten ein dynamischer Zähler mitverwendet, der bei jedem neu erzeugten Agenten in diesem Platz erhöht wird. Darüber hinaus gibt es einen Crash-Zähler, der bei jedem Neustart des Systems erhöht wird und dessen Wert ebenfalls in die Bildung der ID einfließt. All diese Komponenten ergeben in der Summe eine sowohl lokal als auch global eindeutige ID.

2.5 Kennzeichnung

In der Praxis reichen die IDs der Agenten (siehe 2.4) jedoch oft nicht aus. Wollen z.B. zwei Agenten mit ähnlichen Tätigkeitsbereichen miteinander kommunizieren, so müssen sie jeweils die ID des anderen kennen. Wünschenswert wäre es jedoch, wenn ein Agent einfach sagen könnte: *'ich will am Platz xy mit einem Agenten, der die Tätigkeit ABC ausführt, in Kontakt treten'*. Zu diesem Zweck gibt es in Mole die Kennzeichnungen (engl. **badges**). Ein Agent, der z.B. gerade die Aufgabe ABC ausführt, kann die Kennzeichnung 'bearbeite Aufgabe ABC' aktivieren. Ein Agent kann mehrere Kennzeichnungen gleichzeitig tragen. Außerdem ist es möglich, daß ein Agent Kennzeichnungen an andere Agenten weitergibt, oder daß mehrere Agenten die gleiche Kennzeichnung tragen. Somit kann ein Agent durch ein Tupel <*Platz-Bezeichnung, Kennzeichnungsprädikat*> identifiziert werden. Solch ein Kennzeichnungsprädikat könnte z.B. lauten: *'Aufgabe A' UND 'Aufgabe B' ODER 'Aufgabe C'*. Dem Agenten stehen zur Verwaltung seiner Kennzeichnung(en) zwei Methoden zur Verfügung: `pinOnBadge(badge)` und `pinOffBadge(badge)` . Dabei kann er über `pinOnBadge(badge)` eine Kennzeichnung aktivieren und über `pinOffBadge(badge)` eine Kennzeichnung wieder deaktivieren.

2.6 Sessions

Eine Session definiert eine Kommunikationsbeziehung zwischen zwei Agenten. Um zu kommunizieren müssen zwei Agenten erst eine Session veranlassen. Nachdem die Session eingerichtet ist, können die Agenten kommunizieren und Daten austauschen. Dananch wird die Session beendet. Eine Session kann innerhalb eines Platzes stattfinden oder auch zwischen Agenten in verschiedenen Plätzen. Damit die Agenten ihre Unabhängigkeit wahren, muß jeder Partner dem Aufbau einer Session zustimmen. Ein Partner kann jederzeit einseitig eine Session beenden. Sollte ein Partner zu einem anderen Platz wandern, wird die Session automatisch beendet. Diese Mechanismen (Einwilligung, einseitiger Abbruch, automatischer Abbruch) vermeiden, daß ein Agent durch eine Session gegen seinen Willen aufgehalten wird. Durch eine Session entsteht also für einen Agenten keine gefährliche Falle. Wichtig beim Konzept der Sessions ist, daß sich die Partner kennen. Dies ist beim nachfolgend vorgestellten Konzept des Event-Modells (siehe 2.7) nicht der Fall. Beim Event-Modell spricht man aus diesem Grund auch von anonymer Kommunikation.

Warum Mole das Session-Konzept unterstützt, hat einen einfachen Grund: Sessions können genutzt werden, um Agenten zu synchronisieren. Um eine Session anzustoßen, gibt es für Agenten zwei Methoden: `PassiveSetUp()` und `ActiveSetUp()`. Mit `PassiveSetUp()` gibt der Agent einfach seine Bereitschaft bekannt, eine Session einzugehen. Er wird dadurch nicht blockiert. Durch `ActiveSetUp()` wird er hingegen blockiert, bis die Session erfolgreich eingerichtet ist. Um eine Verklemmung zu vermeiden, gibt es einen timeout-Mechanismus, der nach einer gewissen Zeit das Warten abbricht. Sobald `ActiveSetUp()` erfolgreich war, liefert es eine Referenz auf ein Session-Objekt zurück. Die Parameter von `ActiveSetUp()` sind *PlaceID* und *PeerQualifier*. *PlaceID* gibt den Platz des gewünschten Agenten an. Der *PeerQualifier* kann entweder die ID eines Agenten sein oder eine Kennzeichnung. Sollten mehrere Agenten an diesem Platz die gewünschte Kennzeichnung tragen, so wird einer von ihnen zufällig ausgewählt. Bei der `PassiveSetUp()`- Methode sind beide Parameter optional. Sollten sie nicht angegeben werden, so bedeutet dies, daß der Agent bereit ist, mit jedem beliebigen Agenten an jedem beliebigen Platz zu kommunizieren. Um eine Session zu beenden, kann einer der beiden Agenten explizit `SessionObject.Terminate()` aufrufen. Der Partneragent bekommt das Beenden der Session durch einen Aufruf der Methode `SessionTerminated()` mit. Eine Session kann, wie bereits erwähnt, auch implizit durch das Abwandern eines Agenten beendet werden.

2.7 Das OMG Event-Model

Die *Object Management Group event services specification* [OMG94] definiert einen Ereignisdienst mit Lieferanten und Konsumenten. Lieferanten sind Objekte, die Ereignis-Daten produzieren und diese über den Ereignis-Service (engl. **event service**) anbieten. Ein Konsument muß sich für die Ereignisse, über die er informiert werden will, eintragen. Ein Ereignis ist dabei ein Objekt speziellen Typs, das Informationen enthält.

Zwischen den Lieferanten und Konsumenten gibt es zwei Kommunikationsmodelle: das *push*- und das *pull*-Modell. Bei beiden Modellen ist die Kommunikation synchron. Beim *pull*-Modell holen sich die Kosumenten die Daten, indem sie sie explizit beim Lieferanten anfordern. Beim *push*-Modell dagegen sendet der Lieferant die Ereignisdaten an alle registrierten Konsumenten-Objekte. Die Flexibilität dieses Ereignis-Services ergibt sich aus dem Ereigniskanal. Für einen Lieferanten sieht der Ereigniskanal wie ein Konsument aus, aus der Sicht des Komsumenten ist der Ereigniskanal wie ein Lieferant. Dabei können die Lieferanten und Konsumenten frei miteinander verbunden werden. Sie müssen dabei nicht einmal die Identität des jeweils anderen kennen. Aus diesem Grund spricht man auch von anonymer Kommunikation. Die nachfolgende Abbildung 2 soll diesen Umstand verdeutlichen.

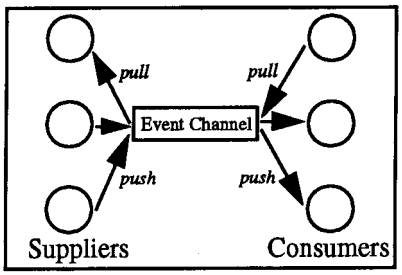


Abbildung 2: der OMG-Ereignis-Kanal (engl. **event channel**) [BHRS98b]

Zwar gibt es mittlerweile Produkte auf dem Markt, die der OMG-Spezifikation folgen, doch keines dieser Produkte unterstützt mobile Teilnehmer. Aus diesem Grund hat die Universität Stuttgart einen Ereignis-Service nach den Richtlinien des OMG-Modells entwickelt, der mobile Teilnehmer unterstützt. Dabei bildet der Ereignis-Service eine hierarchische Struktur, die aus sogenannten *coordinators* und *event demons* besteht. Die *coordinators* stehen stellvertretend für teilnehmende lokale Netzwerke, die *event demons* für die einzelnen Maschinen in den lokalen Netzwerken. Die *coordinators* kommunizieren über TCP/IP. Es ist möglich, *coordinators* hinzuzufügen und zu entfernen. Die *event demons* kommunizieren über *local broadcast*. Die Übergabe von mobilen Teilnehmern wird so durchgeführt, daß eine verläßliche Zustellung von Ereignissen garantiert werden kann.

2.8 Resource Manager

Die Ressourcenkontrolle ist aus zweierlei Gründen für ein System mobiler Agenten wichtig. Erstens ist es für die Abrechnung bei kommerziellen Systemen/Anwendungen wichtig zu wissen, wie viele Ressourcen verbraucht wurden. Zweitens ist es mit Hilfe der Ressourcenkontrolle möglich, feindliche Attacken von Agenten zu erkennen. In Mole werden folgende Ressourcen überwacht: die CPU-Zeit, die lokale Netzwerkkommunikation, die Kommunikation mit ferngsteuerten Netzwerken, die Anzahl der erzeugten Kinder(-agenten) und die komplette Verweildauer der Agenten am Platz. Damit lassen sich die meisten bösartigen Attacken abfangen und es ist möglich, eine vernünftige Berechnung der verbrauchten Ressourcen anzustellen. Lediglich der Speicherverbrauch wird nicht überwacht, obwohl er wichtig ist. Dafür müßte jedoch die JAVA Virtual Machine modifiziert werden, was der Zielsetzung von Mole zuwiderläuft.

2.9 Directory Service

Mole bietet ein lokales Verzeichnis, welches Informationen über Agenten enthält, die einen Service anbieten. Die Information besteht jeweils aus einem String. Ein Agent kann sich selbst beim *Directory Service* anmelden, indem er einen String mit seinem Service übermittelt. Agenten, die einen Service nutzen wollen, können beim lokalen *Directory Service* anfragen und erhalten dann eine Liste mit Agenten, die diesen Service bieten.

2.10 Der Grafische Agentenmonitor

In Mole bietet der grafische Agentenmonitor *Moleview* die Möglichkeit, Plätze zu untersuchen. *Moleview* liefert einen Überblick über die Agenten an diesem Platz und die Nachrichten, die zwischen Agenten gesendet werden.

2.11 Web Integration

Um möglichst einer großen Gruppe von Anwendern offen zu stehen, wurde von Mole eine modifizierte Version entwickelt, die sich als Applet auf jedem JAVA-fähigen Browser ausfüren läßt. Die Browser-Umgebung ist sehr eingeschränkt, was den Zugriff auf das darunterliegende System angeht, und bietet somit gute Voraussetzungen für ein System mobiler Agenten. Der Nutzer kann somit Agenten direkt aus seinem Browser starten und kann sie wandern lassen, indem er eine URL anklickt. Nähere Informationen zu Mole und Quellcode gibt es unter
http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html.

2.12 Agentengruppen

2.12.1 Einführung

Man stelle sich einen Käufer vor, der einen Fernseher erwerben will. Um interessante Angebote herauszufinden, startet er auf seinem PDA (*personal digital assistant*) einen persönlichen Agenten. Der PDA stellt die Schnittstelle zwischen dem Käufer und dem Agentensystem dar. Dem Agenten werden nun die Wünsche des Käufer übergeben. So bevorzugt er Angebote, die detaillierte technische Informationen und Bilder des Gerätes enthalten. Außerdem sollen nur die fünf nähesten Anbieter in Betracht gezogen werden. Zusätzlich gibt der Käufer ein Zeitlimit vor, in dem die Anfrage ausgeführt werden muß. Nun startet der persönliche Agent einen Kaufagenten (A_{tv}), dem er die Wüsche übergibt. Dieser Kaufagent wandert zum nächst besten Verzeichnisservice (D_s). Dort sind die nächsten fünf Verkaufshallen verzeichnet (m_1-m_5). Der Agent schickt nun zu jeder dieser Verkaufshallen einen Agenten (im folgenden Verkaufshallenagenten genannt, im Bild mit A_1-A_5 gekennzeichnet). Diese fünf Agenten werden zu einer Agentengruppe zusammengefaßt (G_m). Wie jede Gruppe besitzen sie eine Gruppenaufgabe und einen eindeutigen Gruppenbezeichner, über den die ganze Gruppe angesprochen werden kann.

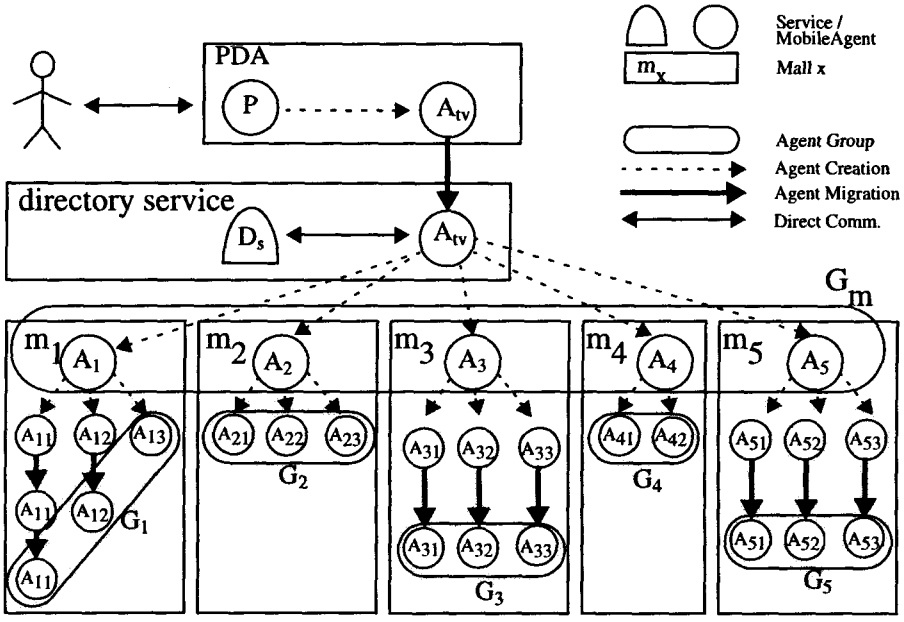


Abbildung 3: Modell eines TV-Kaufs mit Agenten [BaRa]

Wie man dem Bild entnehmen kann, startet jeder Einkaufshallenagent weitere Agenten (z.B. A_{11}), die für einzelne Bereiche der Einkaufshalle zuständig sind. Diese Agenten sind wieder-

um in Gruppen organisiert (z.B. G_1), deren Gruppenaufgabe darin besteht, die besten drei Angebote zu finden.

Die nachfolgenden Kapitel sollen erläutern, aus welchen Elementen solch eine Gruppe besteht und worin die Vorteile liegen.

2.12.2 Das Gruppenmodell

Wie in der nachfolgenden Abbildung 4 dargestellt, besteht eine Agentengruppe aus mehreren Teilen.

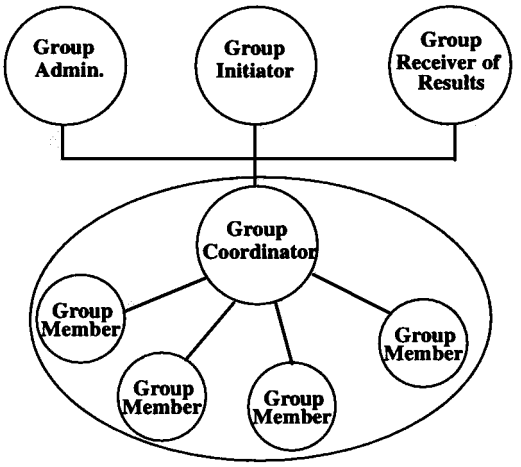


Abbildung 4: das Gruppenmodell [BaRa]

2.12.3 Der Gruppeninitiator

Der Gruppeninitiator ruft die Gruppe ins Leben. Dazu gehört es, Agenten 'einzuladen' und sie der Gruppe zuzuweisen. Außerdem legt der Gruppeninitiator den Gruppenkoordinator, den Gruppenadminstrator und den Empfänger der Gruppenergebnisse fest. Interessant ist, daß der Gruppeninitiator nicht unbedingt der Empfänger der Ergebnisse sein muß.

2.12.4 Gruppenmitglieder

Ein Agent, der an einer Gruppe teilnimmt, ist ein Gruppenmitglied. Da die Kommunikation zwischen den Gruppenmitgliedern anonym verläuft, kennt ein Mitglied die anderen Mitglieder nicht. Ein Gruppenmitglied ist lediglich mit dem Gruppenkoordinator verbunden. Über ihn läuft sämtliche Kommunikation mit den anderen Mitgliedern ab. In dem einleitenden Beispiel tauscht das Gruppenmitglied A_{11} z.B. die drei günstigsten Preise mit den anderen Mitglieder aus. Interessant ist dabei, daß die zentralistische Vorstellung des Gruppenkoordinators nur zur Veranschaulichung des Konzepts dient. In Wirklichkeit besitzt jedes Gruppenmitglied die Gruppenkoordinator-Funktionalität. Diese Realisierung erhöht die Fehlertoleranz, vereinfacht die Verbreitung von Events und reduziert die Kommunikationskosten.

2.12.5 Gruppenkoordinator

Der Gruppenkoordinator ist einerseits für die Koordination innerhalb der Gruppe und andererseits auch für die Koordination der Gruppe mit der Außenwelt verantwortlich. Der Gruppenkoordinator besteht aus einem Bedingungsteil und einem Ausführungsteil. Wenn eine Bedingung

wahr wird, wenn also z.B. ein bestimmtes Event von einem Gruppenmitglied eingetroffen ist, dann wird der entspechende Ausführungsteil ausgeführt. Dies sind zumeist einfache Befehle, wie *'sende Ausgangs-Event', 'ändere internen Zustand', 'stoppe die Ausführung weiterer eintreffender Events'*. Dabei kann der Gruppenkoordinator interne Events ausgeben, die über den internen Ereigniskanal an die Gruppenmitglieder geleitet werden, oder er kann externe Events ausgeben, die über den externen Ereigniskanal der Gruppe an den Gruppenadministrator und die Empfänger der Gruppe gehen.

Auf das Beispiel übertragen bedeutet dies für die Agentengruppe in der Einkaufshalle: Der Gruppenkoordinator erhält alle gefundenen Preise von den Agenten. Ist ein neuer gefundener Preis höher als die besten drei bisherigen Preise, verhält sich der Gruppenkoordinator still. Ist der Preis jedoch niedriger (-> Bedingung erfüllt), so schickt er ein internes Ereignis an alle Gruppenmitglieder, damit diese ihre eigenen Listen aktualisieren können. Wenn nun die Suche fertig ist (z.B. weil das Zeitlimit abgelaufen ist) schickt der Gruppenkoordinator ein Terminierungssignal über den internen Ereigniskanal und verschickt außerdem ein Ausgabe-Ereignis über den externen Ereigniskanal.

2.12.6 Gruppenadministrator

Beim Kreieren der Gruppe legt der Gruppeninitiator einen Gruppenadminstrator fest, indem er dem Gruppenkoordinator einen externen Ereigniskanal anfügt. Dies bedeutet, daß der Gruppenkoordinator nur den externen Kanal kennt und nicht den eigentlichen Gruppenadministrator. Der Gruppenadministrator bestimmt die Lebenszeit der Agenten und ihre Terminierung. Durch das flexible Konzept des externen Kanals ist es möglich, daß der Gruppenadministrator selbst eine Gruppe ist. Dies alles macht das Konzept enorm mächtig. So ist es z.B. möglich, der Gruppe, die für die Administration zuständig ist, neue Mitglieder hinzuzufügen oder Mitglieder zu löschen. Somit kann die Fehlertoleranz des System flexibel gehandhabt werden. Um eine hohe Fehlertoleranz zu erreichen, werden einfach weitere Agenten der Administratorgruppe hinzugefügt. Der Programmierer kann also den Mittelweg zwischen Fehlertoleranz und renundanter Kommunikation selbst festlegen.

2.12.7 Empfänger des Ergebnisses

Der Empfänger des Ergebnisses wird dem Gruppenkoordinator ebenfalls nur indirekt durch den externen Ereigniskanal bekannt gemacht. Somit bietet sich die selbe Mächtigkeit wie bereits beim Gruppenadministrator erwähnt. Der Empfänger des Ergebnisses ist, wie der Name bereits verrät, für den Empfang des Gruppenergebnisses zuständig.

2.12.8 Vorteile des Gruppenkonzeptes am Einkaufs-Beispiel

Wenn man einen Einkaufshallenagenten betrachtet (z.B. A_1), so besteht die Gefahr, daß er durch einen Netzwerkfehler von seiner Einkaufsgruppe (G_1) getrennt wird. Dies hätte zur Folge, daß er das Ergebnis der Gruppe nicht erhält und daß die Gruppe u.U. nicht terminiert. Um dieses Problem zu umgehen, kann man sich das Gruppensystem zu Nutzen machen: Man faßt einfach alle Einkaufshallenagenten (A_1 bis A_5) zu einer großen Gruppe (G_m) zusammen. Wie in den vorherigen Kapiteln erwähnt, ist jedes Gruppenmitglied zugleich auch Gruppenkoordinator. Somit ist die Anzahl der Verbindungen, die verloren gehen müssen, um die Kommunikation völlig zu unterbrechen, deutlich höher (selbst vier Ausfälle führen noch nicht zur Unterbrechung). Allerdings steigen damit auch die Kommunikationskosten unnötig an, da nun die Ergebnisse der Gruppe G_1 unnötigerweise auch an die Gruppenmitglieder A_2-A_5 weitergeleitet werden. Doch erhöhte Fehlertoleranz läßt sich eben nur durch größeren Kommunikationsaufwand bewerkstelligen.

2.13 Sicherheit von Agentensystemen

Bevor im kommerziellen Bereich den Agentensystemen der große Durchbruch gelingt, sind noch einige Sicherheitsprobleme zu lösen. Diese Probleme sind vielfältiger Art. Es gilt, die Agenten gegenseitig zu schützen, den Agenten und den Host voreinander zu schützen, die Hosts untereinander zu schützen und Hosts vor unerlaubten Zugriffen von Dritten zu schützen. Dabei lassen sich durch Kryptographie fast alle Schutzaufgaben erfüllen, lediglich der Agent-Host-Schutz gestaltet sich deutlich schwieriger.

Dies liegt daran, daß der Schutz von Agent und Host zweiseitig ist: der Agent muß vor bösartigen Hosts geschützt werden und der Host muß vor bösartigen Agenten geschützt werden. Diese beiden Aufgaben sollen in den nachfolgenden Abschnitten kurz aufgegriffen werden.

2.13.1 Schutz des Hosts vor den Agenten

Um den Host vor bösartigen Agenten zu schützten bietet sich das Sandbox-Modell an. Diese Technik kommt bereits in verschiedenen Umgebungen, z.B. Java Applets, zum Einsatz. Beim Sandbox-Sicherheitsmodell handelt es sich um um eine geschlossene Umgebung. Innerhalb der Sandbox sind nur Funktionen verfügbar, die kein Sicherheitsrisiko darstellen. Dazu gehören z.B. Addition oder Vergleichsoperationen. Sicherheitsrelevante Funktionen, wie z.B. Zugriffe auf das Dateisystem, sind nicht oder nur unter strenger Kontrolle möglich. Durch dieses Vorgehen läßt sich ein Host recht gut gegen bösartige Agenten schützen. Für einen Host besteht darüber hinaus die Möglichkeit, die Zuwanderung bösartiger Agenten (sofern im vorhinein bekannt) abzulehnen. Anhand der ID eines Agenten kann der Host sofort erkennen, ob der Agent auf einem unsicheren Platz erzeugt wurde und die Annahme des Agenten verweigern.

2.13.2 Schutz des Agenten vor dem Host

Dieses Art des Schutzes ist der eigentlich 'Knackpunkt'. Es ist überaus schwierig, Programme vor ihrer Laufzeitumgebung zu schützen. Jedoch ist gerade dieser Schutz für den Einsatz in kommerziellen Systemen enorm wichtig. Man stelle sich Einkaufsagenten vor, die nach günstigen Angeboten suchen. Dabei legt jeder Agent in seinem Datenteil das bisher günstigste Angebot und den dazu passenden Anbieter ab. Ein böswilliger Host könnte nun u.U. diesen Datenteil verändern und sich selbst als günstigsten Anbieter eintragen. Eine Vielzahl von Manipulationen ist denkbar.

Dabei kann das Vorgehen des Hosts verschiedenartig sein. Er kann

- Programmcode ausspionieren
- Daten ausspionieren
- den Kontrollfluß ausspionieren
- den Code manipulieren
- die Daten manipulieren
- den Kontrollfluß verändern
- den Code inkorrekt ausführen
- sich als anderen Host ausgeben
- eine Ausführung von Code verweigern

- die Kommunikation mit anderen Agenten ausspionieren
- die Kommunikation mit anderen Agenten manipulieren
- Falsche Ergebnisse bei Systemaufrufen zurückliefern

Es gibt bisher vier Ansätze diese Probleme softwaremäßig in den Griff zu bekommen.

Der erste Ansatz beschränkt sich darauf, Hosts als vertrauenswürdig und nicht vertrauenswürdig zu katalogisieren. Den Agenten wird dann nur eine Wanderung zu vertrauenswürdigen Hosts gestattet. Dieser Ansatz löst also nicht das eigentliche Problem, sondern geht ihm aus dem Weg.

Der zweite Ansatz versucht, einzelne Attacken zu erkennen und nachzuweisen. Dem Anwender ist somit der Angriff wenigstens bekannt. Da nur einzelne Attacken erkannt werden können, ist auch dieser Ansatz nicht sehr erfolgsversprechend.

Der dritte Ansatz benutzt Verschlüsselung, um den Agenten zu schützen. Dabei wird nur der Datenteil des Agenten verschlüsselt. Da der Angreifer die Daten nicht verständlich lesen kann, ist er auch nicht in der Lage, die Daten sinnvoll zu modifizieren. Dieser Schutz ist günstig, was die Kosten angeht und er ist nicht zeitbeschränkt, was beim vierten Ansatz der Fall ist. Allerdings ist es bei der Kommunikation nur bei Sendungen zu vertrauenswürdigen Hosts möglich, Klartext-Daten zu senden, bei Sendungen zu anderen Hosts muß über eine Verschlüsselung der Daten nachgedacht werden. Hierzu kann asymmetrische Verschlüsselung eingesetzt werden, wie sie z.B. von RSA bekannt ist. Dabei werden zur Ver- und Entschlüsselung unterschiedliche Schlüssel verwendet. Durch die Kenntnis eines Schlüssels, kann der andere nicht mit vertretbarem Aufwand berechnet werden. Deshalb kann einer der beiden Schlüssel öffentlich gemacht werden und der Kommunikationspartner kann darauf zugreifen.

Der vierte Ansatz wird von den Mole-Entwicklern favorisiert. Er wird als 'zeitbeschränkte Blackbox'(engl. **Time Limited Blackbox**) bezeichnet. Dabei wird ein ausführbarer Agent umgewandelt in einen Agenten, der zwar ebenfalls noch ausführbar ist, aber darüber hinaus einige weitere wichtige Eigenschaften aufweist. Diese Agenten werden als Blackbox bezeichnet und sind für eine gewisse Zeit gegen Attacken geschützt. Ihr Code und ihre Daten können innerhalb des Verfallsdatums nicht gelesen oder verändert werden. Dazu wird der ursprüngliche Agent in eine Form gebracht, in der es sehr schwer ist, ihn zu analysieren oder während der Ausführung den Programmfluß zu analysieren. Der Angreifer braucht also eine gewisse Zeitspanne, bis er die Blackbox analysiert hat und angreifen kann. Sobald diese vorgegebene Zeitspanne abgelaufen ist und die Sicherheit des Agenten nicht mehr garantiert werden kann, werden der Agent und seine Daten ungültig.

3 Vorteile, Nachteile und künftige Entwicklungen

3.1 Vorteile von mobilen Agenten

In einem ersten Überblick läßt sich sagen, daß mobile Agenten die Kommunikationskosten reduzieren können, asynchrone Interaktionen verbessern und eine flexiblere Softwaredistribution erlauben. Gerade in den Bereichen Informationssuche, Netzwerkmanagement, Electronic Commerce und Mobile Computing bieten sich durch mobile Agenten interessante neue Ansätze. Auf die einzelnen Vorteile soll nun im Detail eingegangen werden.

3.1.1 Reduzierte Kommunikationskosten

Durch den Einsatz mobiler Agenten sinken die Kommunikationskosten nicht automatisch. Durch geschicktes Anwenden des Agentenkonzeptes lassen sich jedoch zwei Arten von Einsparungen erzielen:

Bei der ersten Art der Einsparung werden weniger Kommunikationsverbindungen benötigt und die gesamte Kommunikation ist geringer. Dies läßt sich erreichen, indem man zwei Agenten, die viel miteinander kommunizieren, an den selben Platz bringt. Natürlich hat man somit die zusätzlichen Kosten für die Migration, doch in vielen Fällen ist es lohnenswert, diese Kosten einmal auf sich zu nehmen.

Die zweite Einsparungsvariante bezieht sich auf die Menge der Daten, die über das Netzwerk transportiert werden. Dies läßt sich bei Client/Server-Architekturen verdeutlichen, bei denen der Client eine Filterfunktion besitzt. Im Normalfall werden die Daten vom Server zum Client übertragen und der Client filtert dann die wirklich benötigten Daten heraus. Der Datenübertragungsaufwand läßt sich in diesem Fall reduzieren, wenn die Filterfunktion bereits auf dem Server ausgeführt wird. Der Filter kann also als Agent implementiert werden und auf den Server wandern. Somit werden über das Netz nur die wirklich benötigten Daten übertragen.

Zusätzlich bieten mobile Agenten eine bessere Skalierbarkeit, um auf dynamische Entwicklungen zu reagieren. Sollten z.B. in dem oben genannten Fall weitere Server mit großen Datenbanken hinzukommen, so läßt sich ein Agentensystem gut daran anpassen. Es ist u.a. denkbar, neben den Filteragenten, die direkt auf den Servern tätig sind, eine weitere 'Schicht' Agenten zwischenzuschalten. Diese Agenten haben die Aufgabe, renundante Daten von verschiedenen Servern abzufangen und damit die Kommunikationskosten klein zu halten. Offensichtlich ist dieses Verfahren recht gut skalierbar, da die Filterung verteilt abläuft (die Rechenleistung für die Filterung muß nicht von einem einzigen System erbracht werden) und nahe an den Daten stattfindet (die Kommunikationskosten steigen nicht wesentlich an).

3.1.2 Asynchrone Interaktionen

Man stelle sich einen Clienten vor, der einen umfangreichen Auftrag an einen Server schickt. Obwohl der Auftrag asynchron durchgeführt werden könnte, muß der Client verfügbar bleiben, um Antworten zu empfangen und auf sie zu reagieren. Wenn man sich nun vorstellt, daß es sich bei dem Clienten um einen mobilen Clienten handelt, kann es teuer sein, die Verbindung zu halten (über ein Funknetz ist es u.U. sogar unmöglich, die Verbindung über einen längeren Zeitraum zu halten).

Mit dem Einsatz eines Agenten, der die Aufgaben des Clienten übernimmt, läßt sich das Problem elegant lösen: der Agent wird ins Netz geschickt, wo er die Aufgabe ausführt, bzw. den Server damit beaufragt. Der Agent kümmert sich um alle Antworten des Servers und wickelt die Aufgabe komplett ab. In der Zwischenzeit braucht der Client natürlich nicht mit dem Netzwerk verbunden zu sein. Später verbindet sich der Client wieder mit dem Netz und 'holt' den Agenten ab. Allerdings ist es hierfür nötig, daß ein Agent im Netz garantiert nicht verloren geht und daß er garantiert nur einmal ausgeführt wird. Leider erfüllt noch kein Agentensystem diese Bedingungen vollständig.

3.1.3 Softwaredistribution auf Anforderung

In heute gängigen Client/Server-Systemen müssen neue Softwaremodule 'von Hand' installiert werden. Dies kann entweder durch den Nutzer oder den Netzwerkadministrator geschehen. Dazu sind detaillierte Kenntnisse über den Zustand des Systems nötig. Die Installation hängt

von bereits installierten Software-Paketen ab. Auch die Art des Rechners und das verwendete Betriebssystem können eine Rolle spielen. Wie bereits in Kapitel 2.3 beschrieben, existiert hierzu das sogenannte **Code on Demand**-Verfahren. Mit diesem Verfahren ist es möglich, dem Clienten nicht nur den Code zu übermitteln, sondern auch einen Agenten, der in der Lage ist, den Code zu installieren. Die plattformunabhängige Sprache JAVA kommt Mole dabei zusätzlich zu Gute, denn es ist möglich, auf jedem System denselben Installationsprozeß zu verwenden. JAVA 'verdeckt' die Unterschiede bei der Ausführung des Codes. Diese Vorgehensweise stellt also eine Vereinfachung für Nutzer und Administrator dar.

3.2 Nachteile von mobilen Agenten

Die Nachteile von mobilen Agenten werden wahrscheinlich erst zu Tage treten, wenn die Systeme eine große Verbreitung gefunden haben und dadurch einem Alltagstest unterzogen worden sind. Als Nachteile lassen sich bisher nur fehlende Eigenschaften, wie z.B. das unvollständige Sicherheitsmodell nennen. Auch sind Fragen der Abrechnung nicht geklärt. Die Prozessorrechenzeit, die ein Agent auf einem fremden System verbraucht, muß schließlich irgendwie vergütet werden. Nachteile, wie z.B. eine langsame Ausführung der Mole-Agenten aufgrund der darunterliegenden JAVA-Plattform, lassen sich nicht den mobilen Agenten anlasten.

3.3 Künftige Entwicklungen

Für Systeme mobiler Agenten gibt es im Moment noch viele Verbesserungsvorschläge. Das Entwicklerteam von Mole will sich in nächster Zeit auf ein Performance-Modell und das Gebiet der kommerziellen Anwendungen konzentrieren.

Das Performance-Modell soll dabei helfen, Kommunikationskosten besser abzuschätzen. Einer der Vorteile der Agenten-Wanderung liegt darin, daß die Kommunikationskosten reduziert werden können, indem das auszuführende Programm zu den Daten hinwandert. Das Performance-Modell soll nun helfen abzuschätzen, wann es sich lohnt, Agentenwanderung einzusetzen, und wann es lohneswerter ist, ferngelenkte Prozeduraufrufe zu verwenden.

Auf dem Gebiet der kommerziellen Anwendungen wird heutzutage ein immer höherer Grad an Robustheit gefordert. Dazu gehört, daß Tasks exakt einmal ausgeführt werden, was heutige Systeme für mobile Agenten noch nicht garantieren können. Daneben ist es wichtig, daß im Fall von Netzwerkstörungen etc. zumindest Teile des Agentenzustandes rekonstruiert werden können. Heutige Systeme, wie z.B. Java Transaction Service, erlauben lediglich den Serverzustand zu rekonstruieren.

Neben diesen Schwerpunkten werden natürlich auch Konzepte wie das Agentengruppen-Modell, die Agentensicherheit und fortgeschrittene Kommunikationsmodelle verbessert.

Literatur

[BaRa] J. Baumann und N. Radouniklis. Agent Groups in Moblie Agent Systems. *H. König, K. Geihs and T. Preuß (eds.): Distributed Applications and Interoperable Systems (DAIS'97)*, S. 74–85.

[BHRS+98a] J. Baumann, F Hohl, K. Rothermel, M. Schwehm und M. Straßer. Mole 3.0: A Middleware for Java-Based Mobile Software Agents. *N. Davies, K. Raymond and J. Seitz (Eds.): Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, The Lake District, England*, September 1998, S. 355–370.

[BHRS98b] J. Baumann, F. Hohl, K. Rothermel und M. Straßer. Mole – Concepts of a Mobile Agent System. *World Wide Web* 1(3), 1998, S. 123–137.

[Hohl] F. Hohl. A Model of Attacks of Malicious Hosts Against Mobile Agents. *4th ECOOP Workshop on Mobile Object Systems (MOS'98): Secure Internet Mobile Computations.*

[OMG94] Common Object Services Specification. OMG Document Number 94-1-1, Marz 1994.

[RoHR] K. Rothermel, F. Hohl und N. Radouniklis. Mobile Agent Systems: What is missing? *H. König, K. Geihs and T. Preuß (Eds.) Distributed Applications and Interoperable Systems (DAIS'97)*, S. 111–124.

Objektlokation für mobile Anwendungen

Hans Peter Gundelwein

Kurzfassung

Um mobile Objekte in einem weltweiten Netzwerk zu lokalisieren, bedarf es eines skalierbaren Objektsuchdienstes. Hierbei kann ein Objekt ein Telefon, ein Computer, ein Notebook oder ein PDA sein, aber auch eine Datei oder ein Programm wie z.B. ein mobiler Agent. In dieser Seminarausarbeitung wird der "Globe Location Service" von Maarten van Steen, Franz J. Hauck, Philip Homburg und Andrew S. Tanenbaum vorgestellt und darauf untersucht, ob dieser sich als globaler Objektsuchdienst eignet.

1 Einleitung

Computernetze gewinnen immer mehr an Bedeutung, werden immer schneller noch größer und komplexer. Immer mehr Benutzer haben Zugang zum sogenannten globalen Information Superhighway. Ein Großteil dieser Benutzer wird in naher Zukunft zunehmend mobil sein und Handys, Notebooks, PDAs o.ä. verwenden. Noch mehr Daten und Dienste werden verfügbar sein und zunehmend unter mehreren Adressen an mehreren unterschiedlichen Orten abrufbar sein, um die durch Abruf und Nutzung entstehende Last zu verteilen. Durch neue Technologien werden nicht nur Personen, sondern auch Programme und Daten zunehmend mobil. So gewinnt das Auffinden von Personen, Daten oder Diensten im globalen Informationsnetzwerk immer mehr an Bedeutung. Eine Person, die eine andere Person auf deren Handy oder PDA ereichen möchte, will dies über den Namen oder eine fest Rufnummer tun können, ohne zu wissen, wo sich die zu erreichende Person gerade aufhält. Ein Benutzer, der eine bestimmte Datei sucht, ist an deren Inhalt interessiert. Für ihn ist aber uninteresant, wo diese Datei physikalisch gespeichert ist, oder welche der eventuell existierenden Kopien für ihn am günstigsten zu erreichen ist. Benötigt wird ein globaler Suchdienst, der für den Benutzer vollkommen transparent erscheint, der lokale Gegebenheiten und Repliken ausnutzt und der mit dem Datenaufkommen eines solchen weltweiten Dienstes zurecht kommt.

Das Auffinden von Objekten in Netzwerken, mobil oder nicht, geschieht traditionell mit Hilfe von Namensdiensten, die einen Namen auf eine oder mehrere mögliche Adressen abbilden, unter der die Objekte letztendlich erreicht werden können. Als Beispiel kann hier ein Telefonbuch dienen, bei dem Namen von Personen auf Telefonnummern abgebildet werden.
Bisher existierende Namensdienste, wie z.B. der DNS *(Domain Name Service)* , erweisen sich jedoch für die kommenden Anforderungen als ungeeignet. Die Benennung von Objekten in der Art, daß vom Namen eines Objektes auf dessen Adresse geschlossen werden kann, an dem das Objekt zu finden ist wie z.B
`ftp://ftp.rz.uni-karlsruhe.de/pub/readme.txt`, gestaltet sich schwierig, wenn das Objekt mobil ist, da es bei einem Ortswechsel implizit seinen Namen ändert.
Andererseits möchte ein mobiler Benutzer, der einen Dienst, wie z.B. einen Drucker mit dem Namen `local://laserprinter` benutzt, automatisch die Adresse des Druckers erhalten, dessen Position dem Benutzer gerade am nächsten ist.

Benötigt wird ein Dienst, der in der Lage ist, die Adresse eines namentlich bekannten Objekts zu finden und der zudem dabei folgende Merkmale aufweist:

- Unterstützung von mobilen Objekten. Der Ortswechsel und damit verbundene Wechsel der Adresse muß innerhalb kurzer Zeit Berücksichtigung finden.
- Beliebig viele unterschiedliche Adressen pro Objekt.
- Ausnutzung von Lokalitäten. Lokale Adressen (lokale Instanzen/Repliken) werden bevorzugt benutzt.
- Skalierbar auf mindestens 10^{12} Objekte. Diese Anzahl ergibt sich aus der Schätzung, daß die Erde eine Bevölkerung von ca. $6 \cdot 10^9$ Menschen besitzt und durchschnittlich 100 bis 1.000 Objekte pro Person zu erwarten sind.
- Hohe Verfügbarkeit
- Fehlertoleranz. Bei versagen einzelner Teile des Ganzen, bleibt der Dienst an sich funktionell und konsistent.

Diese Seminarausarbeitung beschäftigt sich mit dem "Globe Location Lervice" von Maarten van Steen, Franz J. Hauck, Philip Homburg und Andrew S. Tanenbaum. Im folgenden wird ein Überblick über das Design und die Funktion des Globe Location Service gegeben und näher untersucht, inwieweit sich dieser Ansatz für den praktischen Einsatz eignet und umsetzen läßt.

2 Der Globe Location Service

2.1 Aufbau und Funktion

Für den Globe Location Service GLS werden alle Objekte mit einer global eindeutigen Identifikation ausgestattet. Unter Objekten können Personen, Dienste sowie Daten verstanden werden. Diese Identifikation, auch Handle genannt, ist mit einem UUID *(Universal Unique Identifier)* in DCE [RoKF92] vergleichbar. Dieser Handle ist weltweit eindeutig und wird auch nach Löschen des zugehörigen Objekts nicht erneut vergeben.
Der Globe Location Service ist in zwei Schichten unterteilt. Die erste Schicht bildet frei wählbare und vom Menschen lesbare Namen auf die Handle IDs ab. Diese Schicht läßt sich mit bestehender Technologie realisieren. Es ist sogar möglich den DNS zu benutzen, um diese Schicht zu realisieren, indem die Handles in TXT Records eingetragen werden [SHBT98].
Die zweite Schicht bildet die Handles auf die aktuellen Adressen der zugehörigen Objekte ab. Handles können mit mehreren Adressen verknüpft sein. Objekte sind also unter verschiedenen Adressen ansprechbar. Die zugehörigen Adressen eines Handles können beliebig wechseln, um ein Migrieren der Objekte zu ermöglichen. Im Folgenden wird die zweite Schicht näher erläutert.

2.1.1 Struktur

Im Globe Location Service wird eine hierarchische Unterteilung eines Netzwerkes in Zonen zugrunde gelegt. Diese Unterteilung hat nur Bedeutung für den Location Service. Jeder Zone läßt sich eine Informationsinstanz zuordnen. Zur Notation steht $dir(R)$ für die Informationsinstanz, die für die Zone R zuständig ist. Daraus ergibt sich für den Globe Location Service eine Baumstruktur, dessen Knoten aus einzelnen Informationsinstanzen gebildet wird. Siehe hierzu Abbildung 1. Die Zonen, die einem Blattknoten des Baumes zugeordnet sind, heißen analog Blattzonen.

Die einzelnen Knoten des GLS sind zuständig für die ihnen zugeordneten Zonen, in denen sich Objekte befinden können. Die Wurzel des Baumes ist für alle existierenden Zonen zuständig.

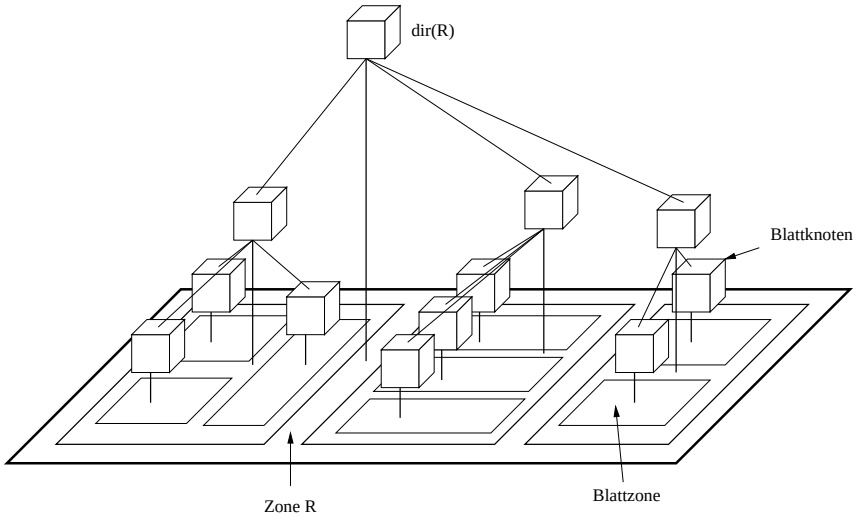


Abbildung 1: Logischer Aufbau des Location Service als Suchbaum

Die Kinder der Knoten unterteilen die Zonen in weitere Subzonen und sind jeweils für diese verantwortlich. Die Knoten besitzen Informationen über alle bekannten Objekte, die sich innerhalb der Zone eines einzelnen Knoten befinden.

Beispiel: Bei Benutzung einer traditionellen Einteilung der Welt, ist der Wurzelknoten zuständig für die gesamte Erde. Er hat Kinder, welche jeweils für einen der fünf Kontinente zuständig sind. Der Knoten für Europa hat Kinder für alle Länder in Europa. Der Knoten für Deutschland hat wiederum Kinder für alle Bundesländer usw.

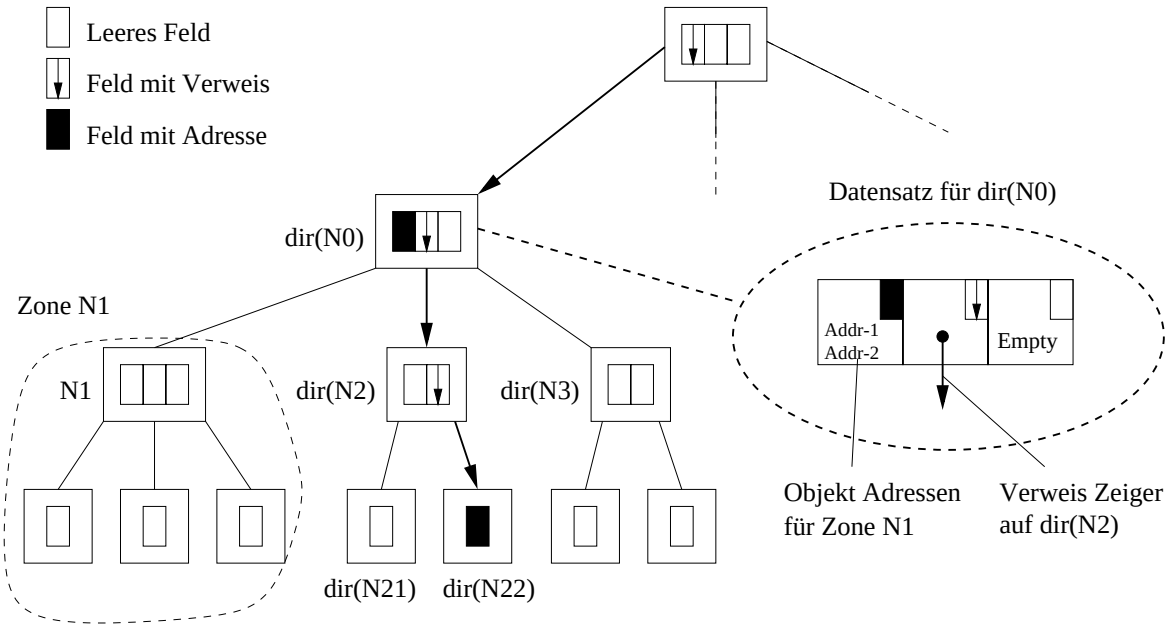


Abbildung 2: Datensätze eines Objektes im Suchbaum

Knoten besitzen für jedes Objekt innerhalb ihrer zuständigen Zone einen Datensatz. Siehe Abbildung 2. Dieser Datensatz enthält entweder eine oder mehrere Adressen eines Objektes oder aber einen oder mehrere Verweise auf andere Knoten, welche die gesuchte Information enthalten könnten. Ein Datensatz enthält entweder Adressen oder Verweise auf andere Knoten, nie aber beides zur gleichen Zeit.

Adressen sind positionsabhängig. Die Zone, in der eine Adresse liegt, ist in der Adresse selbst codiert.

Ausgehend von der Wurzel des Baumes kann die Adresse jedes eingetragenen Objektes gefunden werden, in dem die einzelnen Verweise abgelaufen werden, bis man auf eine oder mehrere Adressen stößt. Verweise sind alle gleichberechtigt. Wenn ein Knoten mehrere Verweise auf ein Objekt enthält, wird ein beliebiger der vorliegenden Verweise benutzt. Um alle Adressen eines Objektes zu finden, müssen jeweils alle Verweise auf das Objekt abgelaufen werden.

Ein mobiles Objekt ändert seine Adresse, wenn es sich von einer Zone in eine andere bewegt, da Adressen zonengebunden sind. Im GLS geschieht dies, indem die neue Adresse des Objekts neu eingetragen und die alte ungültige Adresse des Objektes gelöscht wird.

2.1.2 Hinzufügen neuer Adressen

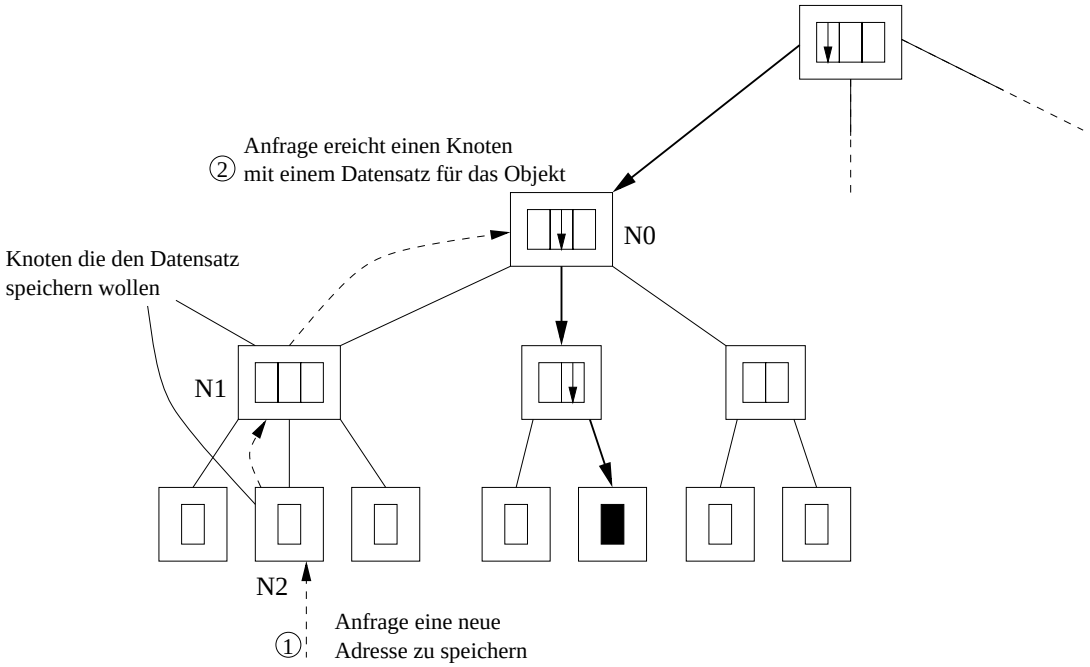


Abbildung 3: Suchbaum während des Einfügens einer neuen Objektadresse

Das Einfügen eines neuen Objektes oder einer neuen Adresse eines Objektes, beginnt am zuständigen Blattknoten der Zone, in der das Objekt mit Adresse eingetragen werden soll. Siehe Abbildung 3. Zunächst wird die Adresse im Blattknoten temporär gespeichert. Danach wird der Vater des Knoten davon informiert und diese Information dem Baum nach oben weiter gegeben. Dabei wird in jedem Knoten, sofern noch nicht vorhanden, ein temporärer Datensatz für das Objekt erzeugt. Die Informationen über die neue Adresse wird entweder bis zur Wurzel weiter gegeben, oder bis zu einem Knoten, der bereits Informationen zu diesem Objekt, in Form von Adressen oder Verweise, gespeichert hat. Ausgehend von dem wurzelnächsten Knoten, der die neue Information erhalten hat, entscheidet jeder Knoten, ob er selbst die Adresse des Objekts speichert oder ob er nur einen Verweis auf das Objekt speichert und seinen Nachkommen erlaubt die Adresse zu speichern. Abbildung 4. Im Normalfall wird die Adresse eines Objekts im Blattknoten gespeichert. Auf jeden Fall existiert für jedes registrierte Objekt, ausgehend von der Wurzel, ein Pfad von Verweisen, der letztlich zu dem Knoten führt, der die Adresse des Objekts gespeichert hat. Sollte während des Einfügens auf einen Knoten gestoßen werden, der schon eine Adresse dieses Objektes abgelegt hat, so wird lediglich die neue Adresse hinzugefügt und die temporären Informationen in den anderen Knoten gelöscht..

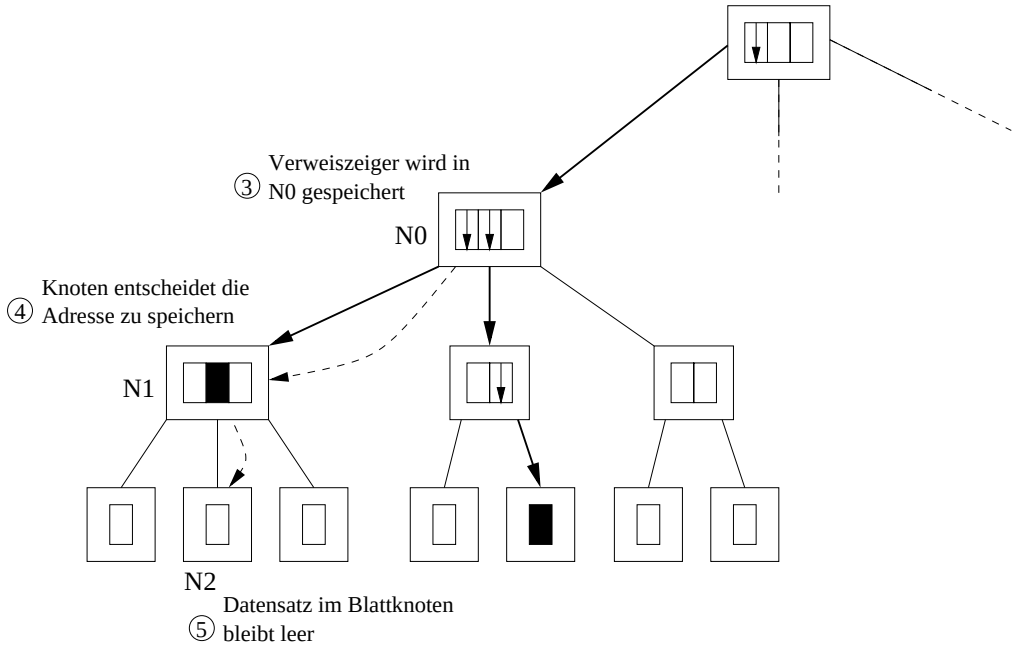


Abbildung 4: Suchbaum nach Einfügen der neuen Objektadresse

2.1.3 Suchen von Adressen

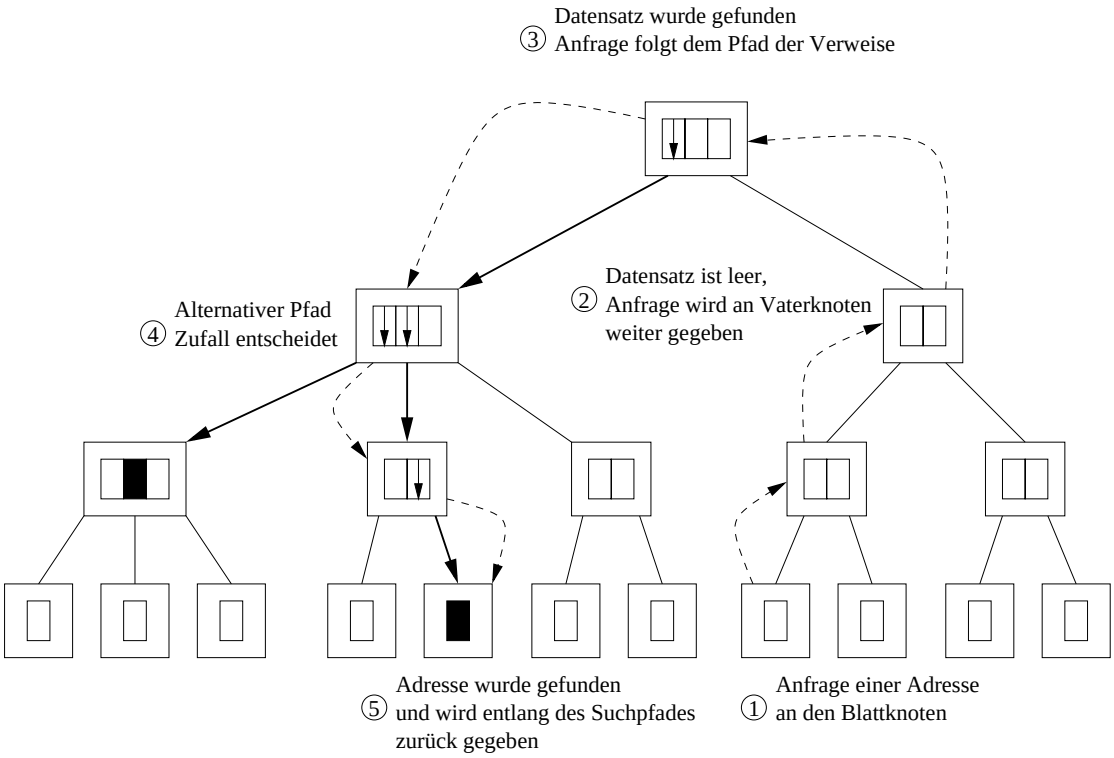


Abbildung 5: Suche einer Objektadresse im Suchbaum

Um die Adresse eines Objektes zu finden, wird eine Anfrage an den Blattknoten der zuständigen Zone gestellt. Abbildung 5. Besitzt dieser Knoten keine Informationen über das gesuchte Objekt, wird die Anfrage rekursiv an den Vaterknoten weitergegeben, im Extremfall bis zum Wurzelknoten des Baumes. Besitzt ein Knoten Informationen über das gesuchte Objekt in Form eines Verweises, wird die Anfrage direkt an den Knoten weitergeleitet, auf den der

Verweis zeigt. Enthält ein Knoten mehrere Verweise auf Adressen des Objektes, so wird ein Verweis zufällig ausgewählt. Besitzt ein Knoten eine oder mehrere Adressen eines Objektes, ist die Suche erfolgreich beendet. Die gefundenen Adressen werden entlang des benutzten Suchpfades zurückgegeben. Sollen alle Adressen eines Objektes gesucht werden, so müssen in einer Tiefensuche jeweils alle Verweise benutzt werden.

Da eine Suche jeweils in einem Blattknoten beginnt, wird zunächst nur nach lokalen Adressen des Objektes gesucht. Existiert lokal keine Adresse des gesuchten Objektes, so wird die Suche Schritt für Schritt weiter ausgedehnt. Dadurch werden bevorzugt nahe Adressen eines Objektes zurückgegeben.

Das Suchen von Adressen läßt sich optimieren, wie in Abschnitt 2.2.2 gezeigt wird.

2.1.4 Löschen von Adressen

Zum Löschen einer Adresse eines Objektes wird zunächst der Knoten gesucht, in dem die Adresse gespeichert ist. Die zu löschende Adresse des Objektes gibt implizit die Zone vor, in dem die Adresse liegt und damit auch den zuständigen Blattknoten. Ausgehend von diesem Knoten wird nach oben zur Wurzel hin nach dem Knoten gesucht, in welchem die zu löschende Adresse gespeichert ist. Nach Auffinden des entsprechenden Knotens, wird die Adresse aus dem Datensatz des Objektes aus diesem Knoten gelöscht. Ist daraufhin der Datensatz des Objektes in diesem Knoten leer, so wird der Datensatz selbst gelöscht und dies dem Vaterknoten mitgeteilt. Daraus ergibt sich eine rekursive Löschung aller Datensätze des Objektes, bis entweder die Wurzel oder ein Knoten erreicht wurde, der nach Löschung des entsprechenden Verweises noch einen oder mehrere Verweise auf dieses Objekt enthält.

2.2 Details

Bisher wurde die grundlegende Funktion des Globe Location Services erläutert. Nachfolgend werden Verfeinerungen beschrieben, welche die Algorithmen schneller machen sollen und dafür sorgen, daß das Konzept skalierbar ist.

2.2.1 Stabile Positionen der Daten

Wie in 2.1.1 beschrieben, bewegt sich ein mobiles Objekt von einer Zone in eine andere, in dem es in der neuen Zone eine neue Adresse übernimmt und seine alte Adresse aufgibt. Wird diese Adresse im Blattknoten der jeweiligen Zone gespeichert, so wird dabei jeweils ein Datensatz mit zugehörigem Datenpfad gelöscht und neu angelegt. Objekte die sich häufig und/oder schnell bewegen, erzeugen so einen relativ großen Verwaltungsaufwand. Um diesen zu reduzieren und um die in 2.2.2 beschriebene Cachestrategie effizient zu machen, wird zur Speicherung der Adresse(n) eines Objektes eine möglichst stabile Position angestrebt.

In Abbildung 6 sieht man eine Zone R mit den Subzonen T_1 bis T_4 und den dazugehörigen Knoten des GLS. Ein Objekt, welches von T_2 nach T_3 wechselt, wird in $dir(T_3)$ eingetragen und in $dir(T_2)$ gelöscht. Jeweils der Verweispfad muß von $dir(R)$ zu $dir(T_2)$ gelöscht bzw. von $dir(R)$ zu $dir(T_3)$ neu aufgebaut werden. Wechselt ein Objekt häufig zwischen T_2 und T_3 hin und her, ist es vorteilhafter, die Adresse des Objektes in $dir(R)$ zu speichern. R umfasst sowohl T_2 als auch T_3. Bei einem Wechsel des Objektes von T_2 nach T_3 oder umgekehrt wird lediglich eine neue Adresse in $dir(R)$ eingetragen. Dies reduziert den Verwaltungsaufwand und stabilisiert die Position, an der die Adresse des Objektes gespeichert ist.

Um entscheiden zu können, wann die Adresse eines Objekts im Suchbaum eine oder mehrere Ebenen näher zur Wurzel abgelegt werden soll, ist es notwendig, jeweils den Zeitpunkt

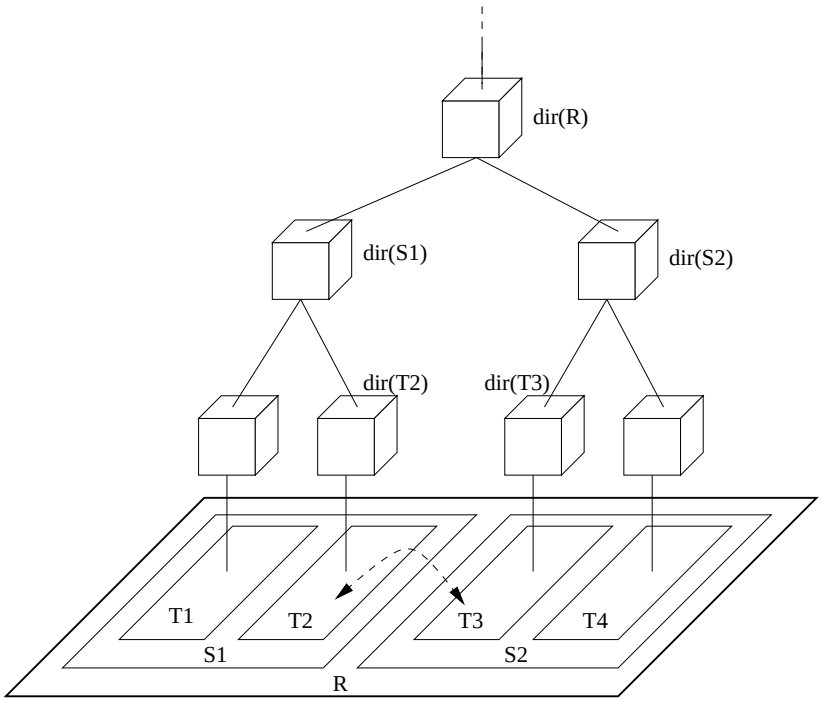


Abbildung 6: Objekt bewegt sich zwischen den Zonen T3 und T4

festzuhalten, an dem ein Verweis neu eingetragen bzw. gelöscht wurde. Hierzu ist es nicht nur notwendig, den Objektdatensatz in einem Knoten um ein Zeitfeld zu erweitern, vielmehr muß diese Zeitinformation auch eine gewisse Zeit im Knoten gespeichert bleiben, nachdem der Datensatz gelöscht wurde.

Hauck, Homburg und Tanenbaum geben in den vorliegenden Publikationen jedoch keinen Algorithmus und keine Methode an, nach der konkret entschieden werden kann, wann und wo eine Adresse eine Objektes gespeichert wird.

2.2.2 Zeiger Caches

Das Suchen von Adressen von Objekten läßt sich erheblich beschleunigen, wenn man in den einzelnen Knoten Caches für Verweise einführt. Hierzu wird nach erfolgreicher Suche, wie in Abschnitt 2.1.3 beschrieben, außer der Adresse des Objektes auch die Adresse des Knoten zurückgegeben, in dem die gefundene Adresse gespeichert ist. Alle Knoten, durch die diese Information durchgereicht wird, sind nun in der Lage, einen Verweis für dieses Objekt in ihrem Cache aufzunehmen, der direkt auf den Knoten zeigt, in dem eine Adresse des Objekts gespeichert ist. Bei erneuter Anfrage nach diesem Objekt kann der befragte Knoten die Anfrage direkt an den Knoten weiterleiten, in dem die Adresse gespeichert ist. Siehe Abbildung 7. So wird die aufwendigere Suche über den kompletten Informationspfad des Objektes vermieden.

Die Einträge des Caches brauchen nicht explizit zu altern, sondern werden bei Ungüligkeit gelöscht (Lazy Cacheing). Läßt sich in dem Knoten, auf den ein zwischengespeicherter Verweis zeigt, nicht der gewünschte Eintrag finden, so wird der Verweis aus dem Cache gelöscht und die Suche wird fortgesetzt, indem die Anfrage an den Vaterknoten weitergereicht wird.

Ansonsten lassen sich alle üblichen Verdrängungsstrategien verwenden.

Dieser Cachemechanismus ist sehr effektiv. Die Strategie greift selbst für mobile Objekte, da im GLS zur Speicherung der Adressen möglichst stabile Positionen angestrebt werden.

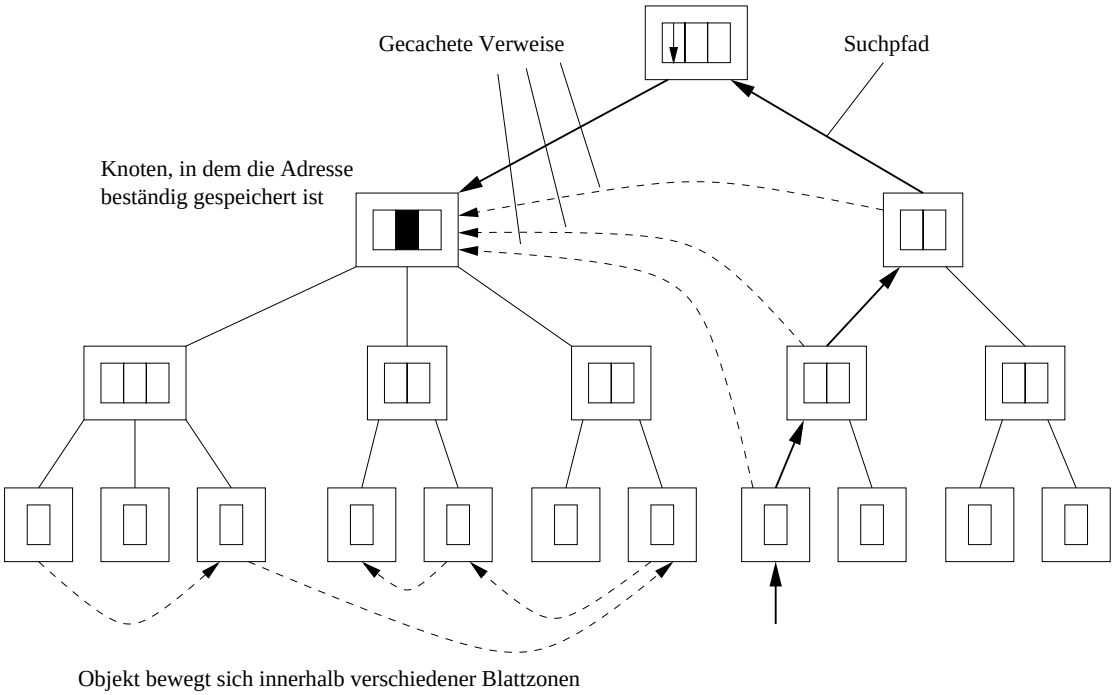


Abbildung 7: Im Cache befindliche Verweise zeigen auf eine beständige Position der gesuchten Adresse, selbst wenn sich das gesuchte Objekt bewegt

Dadurch bleibt der Ort, an dem die Adressen der Objekte gespeichert sind selbst bei starker Bewegung der Objekte relativ stabil.

Diese Cachestrategie in Verbindung zu den stabilen Positionen der Daten wird von den geistigen Vätern des GLS als der Schlüssel dafür angesehen, daß das GLS auch bei großer Skalierung effizient bleibt.

2.2.3 Partitionierung von Knoten

Bis jetzt skaliert der Ansatz des GLS nur schlecht, da in einem einzelnem Knoten je ein Datensatz für alle Objekte benötigt wird, die in der Zone liegen für die der Knoten verantwortlich ist. So existiert eine maximale Größe für eine Zone eines Knotens, bei deren Überschreitung die geforderten Antwortzeiten des Knoten nicht mehr eingehalten werden können, was sich negativ auf den Großteil des gesamten GLS Baumes auswirkt.

Die Lösung besteht in der Partitionierung von zu großen Knoten. Die Idee ist, die ersten n Bits der Handle ID von Objekten zu benutzen, um die Objektdaten auf 2^n Partitionen des Knotens aufzuteilen. Jeder Partition des Knotens werden dabei die Daten der Objekte zugewiesen, deren Handle ID in den ersten n Bits mit der Partitionsnummer übereinstimmt. Diese Lösung besitzt den Vorteil, daß anhand der Handle ID eines Objektes die zuständige Partition eines Knotens bestimmt werden kann.

Die Partitionierung der Knoten stört die beschriebenen Methoden und Algorithmen des GLS nicht. Der Suchbaum bleibt hierdurch konsistent. Die einzelnen Partitionen eines Knoten tauschen untereinander keine Daten aus, da Mengen der Objekte, für die die einzelnen Partitionen zuständig sind, keine gemeinsamen Elemente besitzen und alle Operationen auf jeweils einem Objekt operieren. Kommunikation zwischen Knoten findet nun zwischen den entsprechenden Partitionen statt, ohne die bestehende Ordnung aufzuheben, wie in Abbildung 8 deutlich wird.

Da die Kommunikation zwischen den Knoten nun zwischen Partitionen von Knoten geschieht, ist es notwendig, daß jeder Knoten Informationen darüber besitzt, wie der jeweilige Vater-

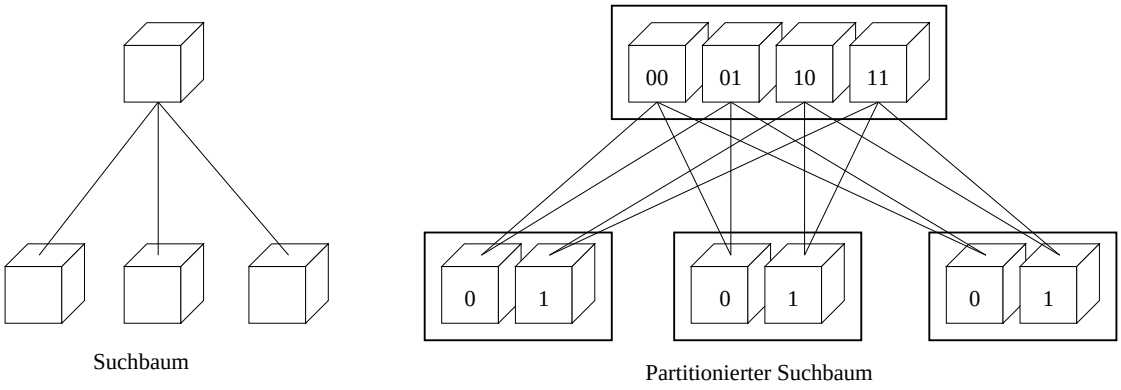


Abbildung 8: Ein Suchbaum und der entsprechende logische Suchbaum nach Partitionierung der Knoten

knoten partitioniert ist. Diese Information ist von einem separaten Baumverwaltungsdienst abrufbar. Diese Baumverwaltung kümmert sich ebenfalls um die Zuteilung der verfügbaren Partitionen auf reale Knoten. Die Partitionierung von Knoten kann als relativ stabil angesehen werden. So kann die Information von den Knoten gecachet werden. Dies ist auch notwendig, da sonst jeder Knoten vor jeder Kommunikation mit anderen Knoten den Baumverwaltungsdienst befragen müßte, was diesen schnell zu einem Flaschenhals machen würde.

2.2.4 Fehlertoleranz

Als potentielles, weit verbreitetes Dienstsystem sollte das Versagen einzelner Stellen bzw. Knoten des GLS, das Funktionieren des Gesamtsystems nicht beeinflussen. Dies ist jedoch nur eingeschränkt möglich, da ein Ausfall eines Knoten den Suchbaum für Stunden teilen kann. In dieser Zeit, bleibt jedoch jeder Teil des Suchbaumes für sich funktionsfähig. Lediglich Informationen aus abgetrennten Zweigen des Baumes sind dann temporär durch die beschriebene Suche nicht auffindbar, wenn diese nicht zwischengespeichert wurden.

Es ist jedoch notwendig zu gewährleisten, daß der Suchbaum auch dann noch konsistent ist, wenn der ausgefallene Knoten, wieder verfügbar ist.

Werden Operationen innerhalb des Suchbaumes in den Knoten in einer Warteschlange gehalten, und werden Anfragen an andere Knoten wiederholt, wenn die entsprechenden Knoten nicht erreicht werden konnten, so gehen dem gesamten Suchbaum keine Informationen verloren und der Suchbaum bleibt konsistent [SHHT98].

Schwierig wird es, wenn die Daten eines Knoten verloren gegangen sind. Es ist jedoch möglich, eine Anfrage an die Kinder des Knotens zu stellen, welche diese dazu veranlaßt dem "verserten" Knoten, all ihre Datensätze mitzuteilen, so daß der Knoten seine Daten teilweise wieder neu aufbauen kann. Adressinformationen von Objekten aus Datensätzen des ausgefallenen Knotens, können jedoch nicht wieder erlangt werden. Daher ist es notwendig einen Knoten mit herkömmlichen Mitteln gegen Datenverlust zu schützen.

3 Skalierbarkeit

Der Globe Location Service ist aufgrund seines verteilten und hierarchischen Konzepts im Prinzip in der Lage, die geforderte Menge an Objekten zu verwalten. Im folgenden soll der Frage nachgegangen werden, wie sich das GLS bei einer derart großen Anzahl von Objekten verhält, und was für Konsequenzen daraus entstehen.

Für das Funktionieren des GLS und dessen Implementierung sind folgende Punkte von Bedeutung, die bisher noch nicht angesprochen wurden:

- Antwortzeiten des GLS auf Benutzeranfragen, bzw. die Zeitspanne, in der Änderungen vollständig bearbeitet werden.
- Benötigte Hardware, sowohl für einen einzelnen Knoten, als auch für das GLS als Ganzes.

Trotz des nachfolgenden Versuchs, diese Punkte zu analysieren, kann endgültige Gewißheit nur nach einem groß angelegten Feldversuch erlangt werden, da bis dahin viele Annahmen getroffen werden müssen, für die bisher die Erfahrungswerte fehlen. Daher werden im folgenden zunächst die Werte und Zusammenhänge analysiert, die sich direkt aus dem Entwurf des GLS ergeben. Danach wird versucht, die angenommenen Kenngrößen so zu wählen, daß diese plausibel sind und das GLS realisierbar bleibt.

Im ungünstigsten Fall muß eine Anfrage an das GLS durch alle Knoten eines Pfades, ausgehend von einem Blattknoten, über die Wurzel zu einem zweiten Blattknoten bearbeitet werden. Der Aufwand für eine Bearbeitung hängt somit von der Tiefe des Suchbaumes ab. Beläuft sich die Anzahl aller Objekte auf l, die durchschnittliche Anzahl von Kindern pro Knoten auf k und die Anzahl der Objekte pro Blattknoten auf o, so ergibt sich eine Tiefe des Suchbaumes von $n = log_k(l) - log_k(o) + 1 = log_k(\frac{l \cdot k}{o})$ wenn dieser, wie wir zu Vereinfachung annehmen wollen, ausbalanciert ist. Eine Anfrage durchläuft im ungünstigsten Fall $2n - 1$ Knoten. Zur Bearbeitung einer Anfrage muß in jedem Knoten der Datensatz des zugehörigen Objektes gefunden werden. Geht man davon aus, daß dies über eine Hashtabelle geschieht, entsteht hierzu ein mittlerer Aufwand von $O(1)$. Die Hashtabelle benötigt im Idealfall als Nettospeicherbedarf einen Zeiger auf den Datensatz des Objekts. Je nach Architektur entspricht das 32 oder 64 Bit wobei 64 Bit für eine zukünftige Implementierung eher wahrscheinlich ist. Suchen im Verweiscache verursacht ebenfalls $O(1)$. Eine Suche nach der Adresse eines Objektes beläuft sich so auf O(1), wenn der Verweiscache greift und $O(n)$ sonst.

Der Eintrag einer neuen Adresse involviert n Knoten. In jedem Knoten wird der Datensatz des Objektes gesucht und falls vorhanden die Informationen (temporär) eingetragen: $O(1)$. Ansonsten wird ein neuer Datensatz erzeugt und in der Hashtabelle eingetragen. Besitzt ein Knoten m Datensätze, bedeutet das einen Aufwand von $O(1)$ oder $O(m)$, wenn die Hashtabelle dadurch zu klein geworden ist.

Da nur erahnt werden kann, nach welchem Algorithmus entschieden wird, wo eine Adresse gespeichert werden soll, wird von einem Aufwand von $O(1)$ ausgegangen. Die Einfügeoperation hat daher im ungünstigsten Fall einen Aufwand von $O(n \cdot m)$, wobei $O(n)$ sehr viel typischer sein wird.

Löschen heißt Finden des Datensatzes und rekursives Löschen des singulären Verweispfades und hat somit ebenfalls einen Aufwand von $O(n)$.

Der Aufwand von Operationen pro Knoten beträgt typischerweise $O(1)$. Da es die Partitionierung von Knoten erlaubt, die Last pro Knoten zu verteilen, und den Speicherbedarf der Objektdaten und Hashtabellen pro System zu reduzieren, kann davon ausgegangen werden, daß durch Ausnutzung der Partitionierung die geforderten Antwortzeiten pro Knoten eingehalten werden können.

Bei einem Aufwand von $O(n) = O(log(\frac{l}{o}))$, oder gar $O(1)$ im GLS Gesamtsystem werden die geforderte Antwortzeiten eingehalten und durch Partitionierung der Knoten mit "Blech erschlagen". Somit ist die Realisierung des GLS eine Frage der Hardware, wie auch im zweiten zu untersuchenden Punkt, dem Speicherbedarf, deutlich wird.

Ein Datensatz eines Objektes besitzt folgenden Aufbau:

1. Die Handle ID des Objektes. [$128Bit$]
2. Ein boolscher Wert zur Unterscheidung, ob der Datensatz Adressen oder Verweise enthält. Theoretisch $1Bit$ praktisch jedoch [$1Byte$]
3. Menge von Verweisen oder Adressen. Die Datenstruktur der Menge selbst benötigt Speicherplatz. Geht man von der platzsparendsten Implementierung aus, einem Array, wird ein zusätzlicher numerischer Wert zur Speicherung der Anzahl der Elemente benötigt. Daher als minimales Datenaufkommen [$1Byte$].
4. Jeder Datensatz enthält entweder Adressen oder Verweise. Beides sind Handle IDs mit einem Speicherbedarf von [$128Bit$]. Der kleinste Datensatz besitzt einen Eintrag.
5. Ein Zeitfeld zum Markieren der letzten Änderung eines Verweiseintrages. Typischerweise derzeit [$4Byte$], die jedoch nur bis ins Jahr 2038 reichen würden.

So ergibt sich ein minimaler Speicherbedarf von 38 Byte pro Datensatz eines Objektes. Zusammen mit dem Speicherbedarf für die Hashtabelle ergibt sich ein minimaler Speicherbedarf pro Objekt von 46 Bytes. Daraus ergibt sich bei 10^{12} Objekten für den Wurzelknoten ein Speicherbedarf von mindestens 41 Tera Bytes ($1\ TeraByte = 2^{40}\ Bytes$). Durch die Möglichkeit, Knoten zu partitionieren, läßt sich sowohl die zu erwartende Last des Knotens, als auch die Datenmenge auf mehrere Systeme verteilen. Bei einer Partitionierung von $p = 8$ ergeben sich $2^p = 256$ Partitionen. Jede Partition hätte dann ein Datenvolumen von 167 GB abzudecken. Auch hier besteht ein Bedarf an sehr viel Hardware. Allein zur Speicherung der 41 TeraBytes an Daten werden ca. 2.000 Festplatten à 20GB benötigt. Ohne Verkabelung, Netzteil und Lüfter entspricht das einer gestapelten Wand von ca. 2m auf 6m. Alternative Speichermedien wie Magneto Optische Platten oder gar Bandlaufwerke sind nicht geeignet, da alle Daten innherhalb kurzer Zeit abrufbar sein müssen.

Für den zentralen Knoten der Welt ist dieser Aufwand sicher gerechtfertigt. Zur Realisierung des GLS sind jedoch viele Knoten notwendig. Alleine der zentrale, für Deutschland zuständige Knoten hat ca. 10^{10} Objekte zu verwalten. Dies entspricht immerhin noch einem Speicherbedarf von mindestens 430GB.

Als realistisch kann man jedoch mit einem Datenvolumen von etwa 100 Bytes pro Datensatz rechnen, was den Speicherbedarf des Wurzelkotens auf 90 TeraBytes und den für Deutschland auf 0,9 TeraBytes ansteigen läßt. Hinzu kommt noch der Speicherbedarf für die Verweiscaches, die bisher nicht berücksichtigt wurden.

Geht man davon aus, daß ein Blattknoten bis zu $o = 10^6$ Objekte verwaltet, so bedarf es für diesen noch ca. 1GB an Speicherplatz. Jedoch werden dann 10^6 Blattknoten benötigt.

Hat ein Knoten im Schnitt $k = 10$ Kinder, so ergibt das einen Suchbaum mit einer Tiefe von $n = log_k(\frac{l \cdot k}{o}) = log_{10}(\frac{10^{12} \cdot 10}{10^6}) = 7$ und über 10^6 Knoten. Jedes Objekt benötigt daher im gesamten Suchbaum in der Regel 7 Datensätze. Dies ergibt ein Gesamtdatenvolumen des GLS von 630 TeraBytes.

Geht man davon aus, daß eine Partition maximal 10^8 Objekte verwalten kann, um die geforderten Antwortzeiten noch einhalten zu können, so kommt man inklusive der Blattknoten auf eine Anzahl von benötigten Systemen von

$$Anzahl_{Sys}(n,k) = \sum_{i=1}^{n-1} k^i \lceil \frac{10^6 \cdot k^{n-i}}{10^8} \rceil = \sum_{i=1}^{n-1} k^i \lceil \frac{k^{n-i}}{10^2} \rceil = \sum_{i=1}^{6} 10^i \lceil 10^{5-i} \rceil$$

$$= 1 \cdot \lceil 10^4 \rceil + 10 \cdot \lceil 10^3 \rceil + 10^2 \cdot \lceil 10^2 \rceil + 10^3 \cdot \lceil 10 \rceil + 10^4 + 10^5 + 10^6 = \underline{\underline{1,15 \cdot 10^6}}$$

Bei 10^6 Systemen für die Blattknoten beträgt der zusätzliche Aufwand des GLS 15%. Geht man jedoch davon aus, daß ein Partitionsystem maximal in der Lage ist, 10^7 Objekte zu verwalten, so ergibt sich für die Anzahl der notwendigen Systeme:

$$Anzahl_{Sys}(n,k) = \sum_{i=1}^{n-1} k^i \lceil \frac{10^6 \cdot k^{n-i}}{10^7} \rceil = \sum_{i=1}^{n-1} k^i \lceil \frac{k^{n-i}}{10} \rceil = \sum_{i=1}^{6} 10^i \lceil 10^{6-i} \rceil$$

$$= 1 \cdot \lceil 10^5 \rceil + 10 \cdot \lceil 10^4 \rceil + 10^2 \cdot \lceil 10^3 \rceil + 10^3 \cdot \lceil 10^2 \rceil + 10^4 \cdot \lceil 10 \rceil + 10^5 + 10^6 = \underline{\underline{1,6 \cdot 10^6}}$$

Dies bedeutet einen Mehraufwand von 40% aufgrund der Struktur des GLS. Das GLS skaliert. Aufgrund der sehr großen Anzahl von zu verwaltenden Objekten ist die benötigte Hardware jedoch enorm.

4 Bewertung

Der Entwurf des Globe Location Services erfüllt alle Forderungen, die an einen globalen Objektsuchdienst gestellt werden. Der Entwurf beinhaltet noch einige Lücken, jedoch sollte es nicht allzu schwierig sein, diese zu schließen. Die Anforderungen an einen globalen Objektsuchdienst, die durch die Anzahl der zu verwaltenden Objekte entsteht, sind enorm. Das GLS ist aber in der Lage, auch diesen Anforderungen nachzukommen. Jedoch verursacht die Struktur der Datenverwaltung des GLS einen erheblichen Mehraufwand an benötigter Hardware. Sollte sich in Probebetrieben herausstellen, daß die angenommenen Werte in Abschnitt 3 zu gering angesetzt sind, besteht die Gefahr, daß das GLS als globaler Dienst nicht mehr realisierbar ist. Verringert sich die Anzahl der maximal verwaltbaren Objekte pro Knoten auf 10^6 oder weniger, so explodiert die Anzahl der benötigten Systeme auf $7 \cdot 10^6$ oder mehr.

Durch Optimierungen der Datenstrukturen besteht ein enormes Einsparungspotential an Hardware. Fest steht jedoch, daß ein immenser flächendeckender Aufwand betrieben werden muß, um das GLS zu realisieren.

Literatur

[RoKF92] W. Rosenberry, D. Kenney und G. Fisher. *Understanding DCE.* O'Reilly & Associates, Sebastopol, CA. 1992.

[SHBT98] Maarten van Steen, Franz J. Hauck, Gerco Ballintijn und Andrew S. Tanenbaum. Algorthmic Design of the Globe Wide-Area Location Service. *Computer Journal* 41(5), 1998.

[SHHT98] Maarten van Steen, Franz J. Hauck, Philip Homburg und Andrew S. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communication Magazine* 36(1), Januar 1998, S. 104–109.

[vSHT96a] Maarten van Steen, Franz J. Hauck und Andrew S. Tanenbaum. A model for worldwide tracking of distributed objects. In *Proc. TINA'96 Conference (Sep. 3-5, 1996, Heidelberg, Germany).* Berlin, Offenbach - VDE, 1996, S. 203–212.

[vSHT96b] Maarten van Steen, Franz J. Hauck und Andrew S. Tanenbaum. A scalable location service for distributed objects. In *Proc. 2nd Annual ASCI Conference (June, 1996, Lommel, Belgium).* Berlin, Offenbach - VDE, 1996, S. 180–185.

Neue Ansätze zur (weiträumigen) Verteilung in vernetzten Systemen

Steffen Schlager

Kurzfassung

Im folgenden werden vier neue Ansätze zur besseren Ausnutzung der vorhandenen Kapazität von verteilten Informationssystemen vorgestellt. *WebWave* [HeMi97], *Nachfragegesteuerte Verbreitung von WWW-Dokumenten* [Best95a] und *Reduzierung der Server-Last und Antwortzeit durch server-seitige Spekulation* [Best95b] versuchen, die vorhandene Last so gleichmäßig wie möglich auf alle Server zu verteilen. In *Effektives kooperatives Caching durch die Verwendung von Hinweisen* [SaHa96] wird versucht, die Leistungsfähigkeit eines kooperativen Caches, der auf mehrere Clientcaches verteilt ist, zu erhöhen. Obwohl es noch keine Versuche in größerem Maßstab gegeben hat, wurde in Simulationen die Effektivität dieser vier genannten Ansätze gezeigt.

1 Einleitung

Die ständig wachsende Menge an verfügbaren Daten in verteilten Informationssystemen, besonders dem WWW, macht es erforderlich, über neue Protokolle zur Verbesserung der Erreichbarkeit von Servern, zur Verkürzung deren Antwortzeiten und zur Verringerung der Netzbelastung nachzudenken. Ein Beispiel für die Notwendigkeit neuer Strategien ist die oft beobachtete Tatsache, daß Server umso schlechter erreichbar sind, desto bekannter und beliebter sie werden. Um diesem Problem entgegenzuwirken, muß sehr viel Wert auf die Skalierbarkeit gelegt werden. Denn ein Protokoll, das in der momentanen Situation zwar funktioniert, aber überfordert ist, sobald sich die Größe des Problems ändert, ist nutzlos angesichts der wachsenden Datenmengen. Um diesem Aspekt gerecht zu werden, versuchen die vorgestellten Ansätze soweit wie möglich auf teuer zu bezahlende globale Informationen und globale Steuerungseinrichtungen zu verzichten, da der dadurch entstehende Kommunikations- und Verwaltungsaufwand die Skalierbarkeit stark einschränkt. Man versucht sich auf lokale, d.h. eventuell ungenaue oder unvollständige Informationen zu beschränken. Dadurch kann es passieren, daß eventuell nicht die richtigen Entscheidungen getroffen werden und daß diese dann korrigiert werden müssen, was natürlich mit Aufwand verbunden ist. Aber wenn die Einsparungen durch den Verzicht auf eine globale Sicht größer sind als die Kosten für die Korrektur falscher Entscheidungen, dann zahlt sich diese Idee aus.

2 Das WebWave-Protokoll

2.1 Einleitung

Das grundsätzliche Ziel des WebWave-Protokolls [HeMi97] ist es, den Durchsatz eines ganzen Netzwerkes zu maximieren. Dazu werden Kopien von *unveränderlichen* Dokumenten (zwecks Aufrechterhaltung der Konsistenz) eines überlasteten Servers auf andere, weniger belastete

Server, verteilt. Sollen auch *veränderliche* Dokumente verteilt werden, dann ist die Konsistenz bzw. Kohärenz der Dokumente zu berücksichtigen. In [HeMi97] geht man der Einfachheit halber aber nur von unveränderlichen Dokuemnten aus. Das Protokoll arbeitet ausschließlich auf der Basis lokaler Informationen und erfüllt deshalb die Forderung nach Skalierbarkeit.

2.2 Lastverteilung

Die Wege, die die Anfragen der Clients an einen bestimmten Server zurücklegen, bilden einen Baum, den sogenannten *Routing-Baum*. Die Wurzel dieses Baumes stellt der Server dar, die Blätter repräsentieren die Clients. Die Idee von WebWave ist nun, daß nicht nur der Server an der Wurzel, an den die Anfrage eigentlich gestellt wurde, diese befriedigen kann, sondern auch jeder Knoten dazwischen. D.h., wenn ein Knoten feststellt, daß er eine Kopie des geforderten Dokumentes besitzt, so kann er dieses Dokument selbst liefern, anstatt die Anfrage zum eigentlich adressierten Server weiterzuleiten. Dabei ist zu berücksichtigen, daß die Knoten zwischen Blätter und Wurzel eigentlich gar keine Server sind, sondern Router, die lediglich für die Wegwahl und nicht für die Lieferung von Dokumenten zuständig sind. WebWave löst dieses Problem dadurch, daß es jedem Router einen Cacheserver zuweist und in die Router Paketfilter einbaut. Entdeckt ein Paketfilter eine Anfrage nach einem Dokument, das sich im zugeordneten Cacheserver befindet, dann wird diese Anfrage abgefangen und das gewünschte Dokument geliefert. Allerdings sollte nicht jede Anfrage auf diese Weise befriedigt werden, selbst wenn das geforderte Dokument vorhanden ist, da sonst das Problem der Überlastung eines Servers nur verschoben, aber nicht gelöst wird.

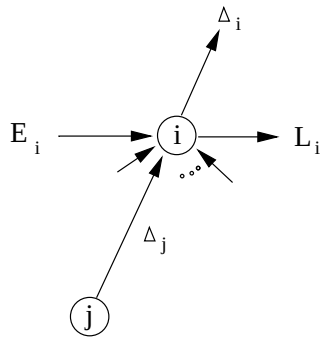


Abbildung 1: Belastung eines Servers im Routing-Baum

Die eigentliche Aufgabe von WebWave liegt deshalb darin, zu bestimmen, wieviel Last einem bestimmten Server zugewiesen wird, d.h. festzulegen, welchen Anteil der Anfragen er selbst bearbeitet und welchen Anteil er nach oben weitergibt. Eine Strategie ist, die Kopien so zu verteilen, daß jeder Server anschließend gleich stark belastet ist. Dieses Vorgehen wird als *Global Load Equalitiy (GLE)* bezeichnet. Allerdings setzt GLE globales Wissen über die Belastung der einzelnen Server voraus. Da aber gerade darauf verzichtet werden soll, kommt dieser Ansatz für WebWave nicht in Frage. Die einzige Information, die WebWave verwenden soll, ist die Belastung der benachbarten Server. Auf die Baumstruktur bezogen bedeutet das, ein Server kennt die Belastung des Vaterknotens und der Kinderknoten.

Abbildung 1 zeigt, welche Last auf den Server i zukommt. Diese setzt sich zusammen aus der *spontanen Anfragerate* E_i und der Summe der weitergegebenen Anfragen der Kinder $\sum \Delta_j$. Server i bearbeitet davon L_i und gibt den Rest Δ_i nach oben weiter. Stellt ein Server nun fest, daß seine Kinder im Vergleich zu ihm weniger belastet sind, dann plaziert er auf diesen Kopien einiger seiner Dokumente, damit Anfragen nach diesen Kopien bereits von den Kindern bearbeitet werden können. Wenn andererseits der Vaterknoten unterbelastet ist, so löschen die Kinderknoten Kopien ihres Vaters, um so Last nach oben weiterzugeben und sich selbst zu

entlasten. WebWave erreicht durch dieses Vorgehen eine Verteilung der zukünftig erwarteten Last eines Servers. Diese Verteilung wird *Tree Load Balance (TLB)* genannt.

2.3 GLE und TLB

Wie schon erwähnt, bedeutet GLE, die Last gleichmäßig auf alle Server zu verteilen. Das wäre zwar z.B. für Prozesse die optimale Verteilung, nicht aber für Dokumente in einem Routing-Baum. Denn werden die Kopien von Dokumenten beliebig auf den Cacheservern verteilt, dann kann es vorkommen, daß viele Anfragen auf dem Weg zum Server an der Wurzel unterwegs nicht auf Kopien der geforderten Dokumente treffen, was ja eigentlich das Ziel von WebWave ist. Also kann so keine Entlastung erreicht werden. Deshalb gibt es beim WebWave-Protokoll sinnvolle Beschränkungen bei der Verteilung der Kopien.

Grundsätzlich soll jeder Server soviel Last wie möglich unter Einhaltung folgender Einschränkungen aufnehmen:

- Der Server an der Wurzel darf keine Anfragen nach oben (wohin auch?) weitergeben, d.h. er muß alle an ihn gestellten Anfragen befriedigen.
- Kopien von Dokumenten werden ausschließlich auf Server plaziert, die sich im Baum abwärts in Richtung der Clients befinden und die dieses Dokument auch angefordert haben. Damit ist sichergestellt, daß diese Kopien auch gefunden werden. Diese Einschränkung bewirkt, daß ein Cache-Verzeichnis, in dem die Inhalte der einzelnen Cacheserver festgehalten werden, nicht mehr nötig ist, um eine bestimmte Kopie zu finden. Abbildung 2 zeigt, wohin Kopien von Dokumenten von Server i verteilt werden dürfen, wenn diese von den Clients j und k angefordert wurden.

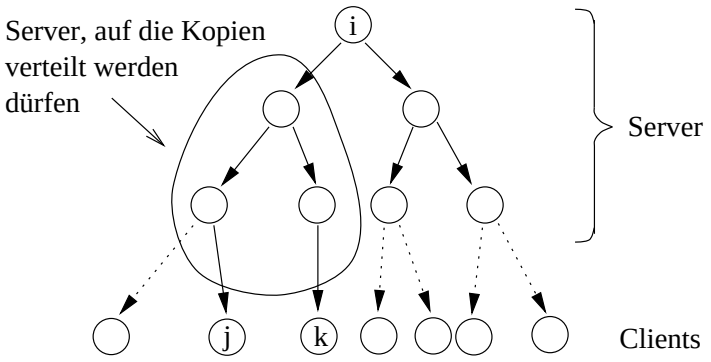


Abbildung 2: Erlaubte Richtungen für die Verteilung von Kopien

Mit diesen Einschränkungen konvergiert die Lastverteilung im allgemeinen jetzt nicht mehr gegen GLE, sondern gegen TLB. Allerdings kann es unter Umständen vorkommen, daß bei einer gewissen Verteilung der spontanen Anfrageraten TLB und GLE zusammenfallen.

2.4 Der Protokoll-Algorithmus

Für den Algorithmus werden folgende Argumente und Parameter verwenden:

- T: Menge der Knoten, die den Routing-Baum bilden.
- α_i: Bruchteil der Überlast, die Server i abgibt.

- C_i: Menge der Kinder von Knoten i
- L_i: Last, die der Server i aufnimmt.
- L_{ij}: Last, die der Server j nach Meinung von Server i aufnimmt. Diesen Wert hat Server i beim letzten Informationsaustausch von Server j erhalten, aber dieser Wert muß nicht immer mit L_j, dem aktuellen Wert, übereinstimmen.

WebWave(T)
$\alpha_i \leftarrow \frac{1}{2+|C_i|}$ /* andere Werte für α_i möglich */
do forever
foreach j child of i in T **do**
/* Lastausgleich mit den Kinderknoten*/
$L'_i \leftarrow L_i - min\{\Delta_j, \alpha_i \cdot (L_i - L_{ij})\}$
for k parent of i **do**
/* Lastausgleich mit dem Vaterknoten */
$L'_i \leftarrow L_i + min\{\Delta_i, \alpha_k \cdot (L_{ik} - L_i)\}$
$L_i \leftarrow L'_i$
send(L_i) to parent(i) and children(i) in T
/* neue Last Vater und Kindern mitteilen */

Dieser TLB-Algorithmus für WebWave läuft perodisch auf jedem Server. Damit schätzen die Server periodisch die Anzahl der zukünftigen Zugriffe ab und verteilen dementsprechend Last an die Kinderknoten und den Vaterknoten. Dieser Algorithmus berechnet lediglich, wieviel Anfragen ein einzelner Server bearbeitet und wieviel er an seinen Vaterknoten weitergibt. Das Plazieren der Kopien auf anderen Servern und die Entscheidung, welche Dokumente kopiert werden, ist hier der Einfachheit halber nicht berücksichtigt.

2.4.1 Mögliche Hindernisse

Da WebWave ausschließlich mit lokalen Informationen arbeitet, kann unter gewissen Umständen eine Situation entstehen, in der die durch WebWave veranlaßte Lastverteilung nicht mehr gegen TLB konvergiert. Diese Situation tritt genau dann ein, wenn ein bestimmter Server zu einer *„Barriere"* wird, die die weitere Verteilung der Kopien in eine gewisse Richtung verhindert. Man bezeichnet einen Server als „Barriere", wenn folgende, in Abbildung 3 gezeigte Situation eintritt:

Der wenig belastete Server S_3 gibt alle Anfragen für ein Dokument d_3 nach oben zum Server S_2 weiter, weil er selbst keine Kopie dieses Dokuments besitzt. Server S_2 sei stark belastet und würde gerne Last an Server S_3 abgeben. Angenommen, S_2 besitzt keine Kopie von Dokument d_3, dann kann S_2 natürlich keine Last auf S_3 verteilen, indem S_2 eine Kopie von d_3 auf S_3 plaziert. Wenn keine anderen Dokumente von S_3 verlangt werden, dann bleibt dieser Server unterbelastet und eine Lastverteilung entsprechend TLB ist nicht mehr möglich.

Um dieses Problem zu lösen, muß der Algorithmus etwas erweitert werden. Wenn ein Server S_j länger als zwei Perioden unterbelastet ist im Vergleich zu seinem Vater S_i, dann nimmt S_j an, daß S_i eine „Barriere" ist. Um dieses Hindernis zu beseitigen, fordert S_j selbst einige der Dokumente an, deren Anfragen er bisher an S_i weitergegeben hatte und plaziert diese Dokumente in seinem Cache. Dieses Methode wird als *„Tunneln"* (engl. tunneling) bezeichnet.

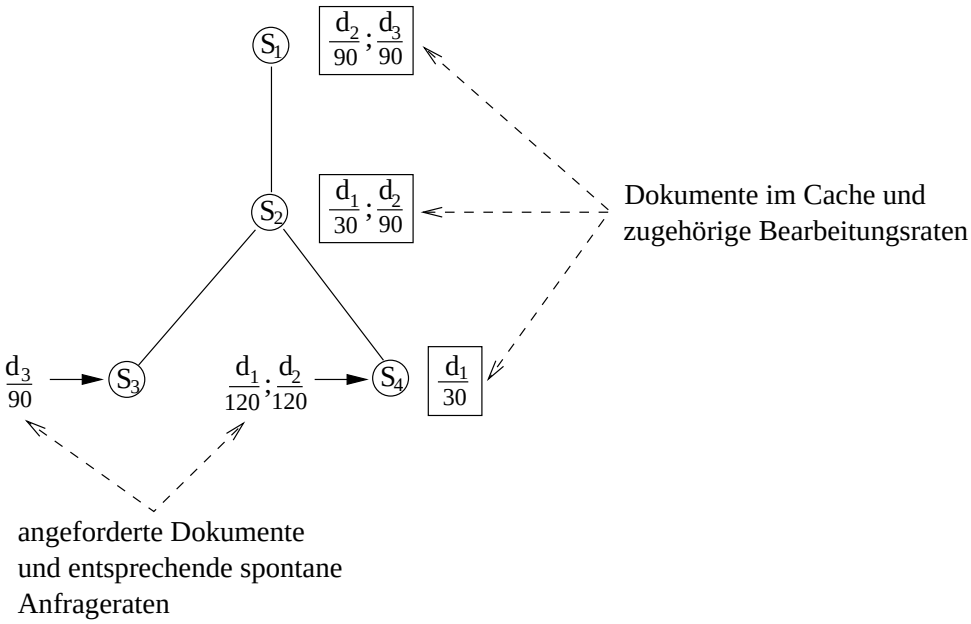


Abbildung 3: Skizze für eine Barriere

3 Nachfragegesteuerte Verbreitung von Dokumenten

3.1 Einleitung

Ähnlich wie bei WebWave, versucht man bei der nachfragegesteuerten Verbreitung von Dokumenten [Best95a] die Last auf viele Server zu verteilen, um dadurch den Netzwerkverkehr zu verringern. Die Idee hierbei ist, daß man versucht, Kopien beliebter und häufig angeforderter Dokumente auf Server, die möglichst nahe beim Client sind, zu plazieren. D.h. je beliebter ein Dokument ist, desto näher soll es beim Kunden sein. Diese Idee der Verteilung ist uns im täglichen Leben nicht fremd: große Firmen besitzen Filialen um näher beim Kunden zu sein. Briefe und Zeitungen werden nicht direkt, sondern über Austräger zugestellt, die jeweils nur für einen relativ kleinen Bereich zuständig sind. Dieses Prinzip hilft, Kosten und Zeit zu sparen. Auf Netzwerke übertragen heißt das Verringerung der Netzbelastung und Verkürzung der Antwortzeit.

3.2 Zugriffsstatistiken für Server

Wenn man die Zugriffsstatistiken eines Servers analysiert, stellt man fest, daß sich die meisten Zugriffe auf eine relativ kleine Anzahl von Dokumenten beschränken. Abbildung 4 zeigt die Anzahl der Zugriffe auf einzelne 256KB-Blöcke mit abnehmender Zugriffshäufigkeit auf den HTTP-Server cs-www.bu.edu der Universität Boston. Dieser Server wird in erster Linie von lokalen Clients kontaktiert. Alleine der am häufigsten angeforderte Block, der 0,5% der verfügbaren Datenmenge ausmacht, ist für 69% der Zugriffe auf den Server verantwortlich. 10% aller Blöcke machten 91% Prozent der Zugriffe aus.
Diese Statistik zeigt, daß sehr viel Bandbreite gespart werden könnte, wenn man Anfragen für beliebte Dokumente schon auf einer früheren Stufe befriedigen könnte. Dies erreicht man durch einen *Serverproxy*, der einer bestimmten Menge von Servern zugeordnet wird, z.B. den Servern innerhalb eines LAN einer Universität. Auf diesem Serverproxy werden die Kopien der beliebtesten Dokumente der einzelnen Server plaziert.

Die Entscheidung, welche Dokumente auf diesem Proxy plaziert werden, hängt aber nicht nur von der Zugriffshäufigkeit ab. Wichtig ist auch, ob auf ein Dokument häufig innerhalb

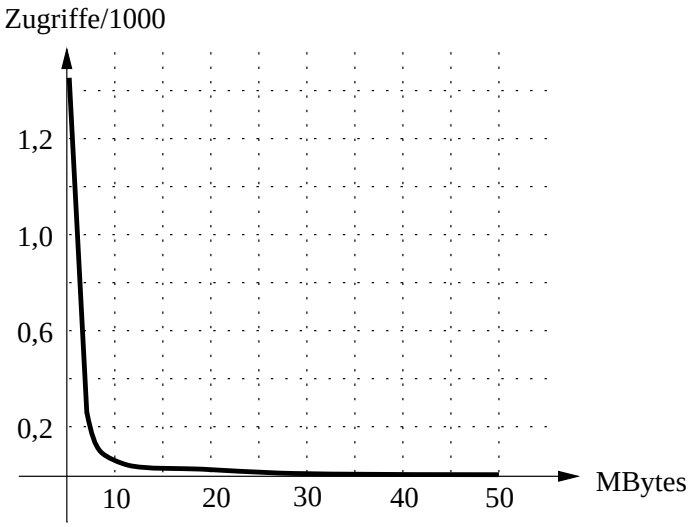


Abbildung 4: Zugriffshäufigkeit auf den HTTP-Server cs-www.bu.edu

eines LAN zugegriffen wird oder in erster Linie von außerhalb und wie häufig ein Dokument geändert wird. Dementsprechend teilt man die Dokumente in folgende Klassen ein:

- *locally popular*, d.h. vor allem Zugriffe innerhalb des LAN
- *remotely popular*, d.h. vor allem Zugriffe von Clients außerhalb des LAN
- *globally popular*, d.h. sowohl Zugriffe von Innen als auch von Außen

Je nachdem, wie oft ein Dokument geändert wird, wird es bei [Best95a] einer der beiden folgenden Klassen zugeordnet:

- *mutable*, d.h. die Dokumente werden ziemlich häufig geändert
- *immutable*, d.h. die Dokumente werden selten geändert

Es ist klar, daß sich vor allem remotely popular/immutable Dokumente dazu eignen, auf einem Proxy plaziert zu werden, denn so werden die Server innerhalb eines LAN weitgehend von Anfragen von außen entlastet. Außerdem ist es nicht nötig, die Dokumente auf dem Proxy häufig zu aktualisieren, da dort vor allem solche Dokumente plaziert werden, die sich selten ändern. Nach [Best95a] reichen für die Erhaltung der Konsistenz angesichts der seltenen Änderungen der Dokumente einfache Protokolle wie *Time-To-Live (TTL)* oder *Alex* [Cate92] aus.

3.3 Systemmodell

Das Systemmodell sieht eine Partitionierung der einzelnen Server in verschiedene Cluster vor. Ein bestimmter Server eines Clusters dient als Proxyserver. Ein Server kann mehreren Clustern angehören und hat dann dementsprechend mehrere Proxies, auf denen Kopien seiner Dokumente abgelegt werden können. Abbildung 5 zeigt ein Beispiel einer Partitionierung.

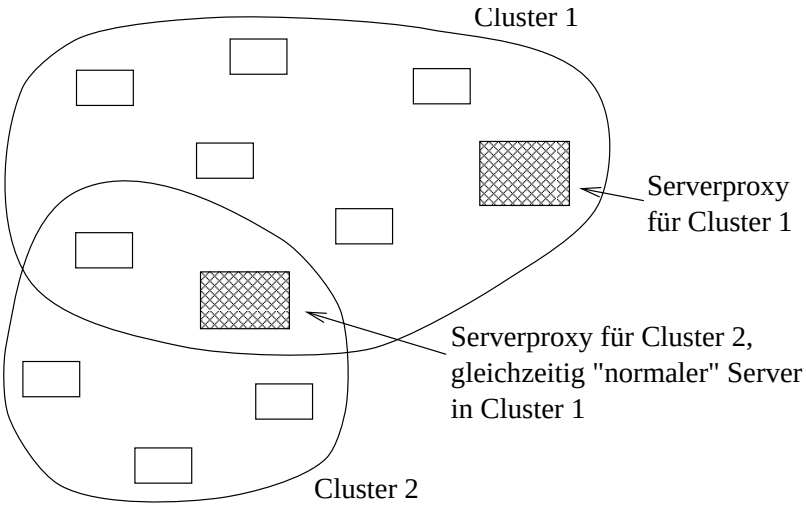


Abbildung 5: Systemmodell: Partitionierung der Server als Cluster

3.4 Das DDD-WWW-Protokoll

Das im folgenden vorgestellte DDD-WWW-Protokoll (DDD steht für „Demand-based Document Dissemination") ist für das WWW entworfen worden und ermöglicht es einem Client, gewünschte Dokumente von einem Server in seiner Nähe zu beziehen.

Ein Server kann leicht eine Statistik über die Zugriffshäufigkeiten der einzelnen Dokumente, die er anbietet, erstellen. Mit Hilfe dieser Statistik entscheidet ein periodisch auf jedem Server ablaufender Algorithmus, welche Dokumente auf den Proxy kopiert werden. Da sich laut Statistik die Zugriffshäufigkeiten nur sehr langsam in der Zeit ändern, muß dieser Algorithmus nicht sehr oft ausgeführt werden. Die Berechnung kann sogar offline erfolgen. In [Best95a] werden Perioden von 30 Tagen als ausreichend angesehen. Ferner hält ein Server fest, welche Dokumente er an welchen Proxy verteilt hat. Wenn ein Client von einem Server nun ein Dokument anfordert, dann bekommt er dieses geliefert, wenn das gewünschte Dokument nicht auf einen Proxy kopiert wurde. Ist dies aber der Fall, dann bekommt der Client die URL des entsprechenden Proxy. Da die Verbreitung der Kopien rekursiv fortgesetzt wird, besteht die Möglichkeit, daß das Dokument nochmals an einen Proxy weitergegeben wurde, der sich noch näher beim Client befindet. Der Client fragt nun diesen Proxy nach dem gewünschten Dokument und dieser antwortet entweder direkt mit dem Dokument oder er gibt wiederum eine URL eines Proxy zurück. Je häufiger man weiterverwiesen wird, desto näher befindet sich der entsprechende Proxy beim Client. Dadurch wird dann bei der tatsächlichen Lieferung des Dokuments sehr viel Bandbreite gespart und die Server, auf denen das Dokument eigentlich abgelegt ist, werden stark entlastet. Allerdings ist die iterative Ermittlung der URL des nächsten Server zeitintensiv. Damit sich dies nicht zu negativ auf die Antwortzeit auswirkt, benötigt jeder Client einen *URL Translation Lookaside Buffer*, in dem die zuletzt verwendeten URLs gepuffert werden. Bei einer Anfrage schaut der Client dann zuerst in seinem Puffer nach, ob die gewünschte URL vorhanden ist. Da bei einem Cache die Trefferraten normalerweise sehr hoch sind, kommen die zeitintensiven URL-Ermittlungen nur selten vor.

Bezogen auf die Zugriffsstatistiken des Servers cs-www.bu.edu werden in [Best95a] folgende Ergebnisse berechnet:

Wenn man 10 derartige Server zu einem cluster zusammenfaßt, dann genügt bereits ein Proxy mit 36MBytes Speicherplatz, um die Bandbreite jedes einzelnen Servers um 90% zu reduzieren. Ein Proxy mit 500MByte ist in der Lage, die Bandbreite von 100 Server um je 96% zu verringern.

Natürlich stellt sich die Frage, ob dieser Proxy, der einen großen Teil der Anfragen befriedigen soll, nicht selbst zu einem Flaschenhals wird. Dies wäre tatsächlich der Fall, wenn die Verbreitung beliebter Dokumente nicht rekursiv auf weitere Proxyserver fortgesetzt wird. Dadurch wird insgesamt eine globale Verteilung der Last erreicht. Wenn eine Weiterverteilung der Last aus irgendeinem Grund nicht mehr möglich ist, dann kann man immer noch die Speicherkapazität des Proxy dynamisch anpassen und so bei großer Belastung durch Verringerung der Kapazität eine Entlastung des Proxy erreichen. In diesem Falle läßt der Proxy mehr Anfragen an die Server innerhalb des Cluster durch.

4 Effektives kooperatives Caching durch die Verwendung von Hinweisen

4.1 Einleitung

Caching ist eine häufig angewandte Technik, um die Leistung von verteilten Dateisystemen zu erhöhen. Das kooperative Caching ist in der Pufferungshierarchie zwischen dem *Clientcaching* und dem *Servercaching* einzuordnen. Clientcaches filtern die Anfragen von Anwendungen, um Netzwerk- und Serverbelastung zu vermeiden. Servercaches filtern Anfragen an den Server, um teure Plattenzugriffe zu verhindern. Die Frage ist, wer den kooperativen Cache kontrolliert, der auf viele Clients verteilt ist. Eine Lösung ist ein globaler Manager. Aber da auf Skalierbarkeit sehr viel Wert gelegt wird, scheidet diese Lösung von vorne herein aus. Eine andere Lösung, die im folgenden vorgestellt wird, beruht auf dem Prinzip, daß jeder Client einige Aufgaben des Managers selbst übernimmt. Allerdings verfügen die Clients nicht über globale Informationen, sondern sie verlassen sich auf *Hinweise*. Simulationen haben gezeigt, daß Hinweise ausreichen, um dieselben Cache-Trefferraten zu erzielen wie bei Verwendung globaler Informationen. Hinweise aber erfordern sehr viel weniger Verwaltungs- und Kommunikationsaufwand, was sich in einer erheblichen Reduzierung des Verschnitts niederschlägt.

4.2 Kooperatives Caching

Findet ein Client ein gesuchtes Dokument nicht in seinem eigenen lokalen Cache, so versucht er, den gewünschten Block im kooperativen Cache zu finden. Dieser Vorgang wird als *„Block Lookup"* bezeichnet. Der Client muß dazu aber wissen, ob sich der gesuchte Block überhaupt im kooperativen Cache befindet und wenn ja, welcher der am kooperativen Cache beteiligten Clients den Block in seinem eigenen Cache hat. Was diese Information angeht, verläßt sich der Client auf seine Hinweise.

Weiterhin spielen beim kooperativen Caching folgende Punkte eine Rolle:

- Die *Ersetzungsstrategie* (replacement policy) bestimmt, was passiert, wenn ein Client einen Block aus seinem eigenen Cache löschen will. Eine Möglichkeit ist, den Block an einen anderen Client weiterzuschicken und ihn so im kooperativen Cache zu behalten. Die andere Möglichkeit ist, den Block einfach zu verwerfen.
- Die *Cache Konsistenz* wird durch Token, die der Manager vergibt, gewährleistet. Ein „Grant Token" erlaubt einem Client den Zugriff auf eine ganze Datei. Ein „Revocation Token" zwingt den Client, seine Kopie einer bestimmten Datei zu löschen. Die Aufrechterhaltung der Konsistenz durch die Vergabe von Token durch den Manager mag zwar immer noch als Flaschenhals erscheinen, aber diese Maßnahme ist unumgänglich. Ferner ist zu beachten, daß ein Client nur einmal den Manager kontaktieren muß, um das Token für den Zugriff auf eine Datei zu erhalten. Danach kann der Client jeden „Block Lookup" für diese Datei ohne Manager mit Hilfe seiner Hinweise durchführen.

4.3 Bisherige Algorithmen

Um nachher den auf Hinweisen basierenden Algorithmus bewerten zu können, werden zum Vergleich zwei bisherige Algorithmen für das kooperative Caching kurz vorgestellt: *N-Chance* [DWAP94] und *GMS (Global Memory Service)* [FMPK+95].

- *N-Chance*:
  Beim N-Chance-Algorithmus wird der lokale Cache des Client dynamisch in zwei Teile aufgeteilt. Der eine Teil enthält die Blöcke des kooperativen Cache, der andere Teil die Blöcke, die der Client selbst benötigt. Der Manager ist für die Aufrechterhaltung der Konsistenz verantwortlich. Außerdem verwaltet er die Information, bei welchem Client welche Blöcke zu finden sind. Als Ersetzungsstrategie wird eine Kombination aus lokalem LRU und der Vermeidung von Duplikaten verwendet. Ein Client löscht bei Bedarf immer seinen lokal ältesten Block. Ob der Block verworfen oder weitergeschickt wird, hängt von der Anzahl der Kopien des betreffenden Blocks ab. Blöcke mit mehr als zwei Kopien werden verworfen und Blöcke mit höchstens einer Kopie werden zufällig an einen anderen Client weitergeschickt. Um herauszufinden, wieviele Kopien von einem bestimmten Block existieren, fragt der Client den Manager.

- *GMS*:
  Bei GMS verwaltet ebenfalls der Manager die Informationen, wo welche Blöcke zu finden sind. Im Unterschied zu N-Chance bietet GMS keinen Konsistenzmechanismus. Die Ersetzungsstrategie vermeidet ebenfalls das Prinzip der Vermeidung von Duplikaten, um zu entscheiden, ob ein Block verworfen oder weitergeschickt wird. Allerdings wird bei GMS ein Block nur dann weitergeschickt, wenn überhaupt keine Kopie des Blockes im kooperativen Cache vorhanden ist. Um herauszufinden, ob Kopien eines Blockes vorhanden sind, schaut der Client ein Tag an, das jedem Block „angeheftet" wird und das diese Information enthält. Der Manager führt Buch über die Anzahl der Kopien und informiert einen Client, wenn dieser einen Block enthält, von dem keine Kopien mehr existieren. Der Client setzt dann den Tag des Blockes auf den entsprechenden Wert. Im Gegensatz zu N-Chance wird im Falle des Weiterschickens eines Blockes das Ziel nicht zufällig bestimmt. Bei GMS überwacht der Manager, welcher Client den global ältesten Block in seinem Cache hat. Diese Information schickt er periodisch an alle Clients, so daß diese wissen, wohin sie einen Block weiterschicken sollen. Dadurch wird ein globales LRU über alle Clientcaches approximiert.

4.4 Der auf Hinweisen basierende Algorithmus

Die bisher existierenden Algorithmen N-Chance und GMS verwenden für einen „Block Lookup" alle den zentralen Manager. Der im folgenden beschriebene Algorithmus verwendet Hinweise, die eventuell auch falsch sein können, was eine Korrektur erforderlich macht. Aber solange die Reduzierung des Verschnitts durch Anfragen an den Manager größer ist als der Korrekturaufwand bei falschen Hinweisen, zahlt sich der Algorithmus aus.
Zwei grundsätzliche Funktionen zeichnen den Algorithmus aus:

- Verwaltung der Hinweise
- „Block Lookup"-Mechanismus

4.4.1 Verwaltung der Hinweise

Wenn ein Client auf eine Datei im kooperativen Cache zugreifen will, muß er vom Manager zuerst das erforderliche „Grant Token" anfordern. Wenn noch kein anderer Client dieses Token besitzt, schickt der Manager dem Client das Token und gleichzeitig eine Liste mit Hinweisen, wo sich wahrscheinlich die sogenannte *„Mastercopy"* eines jeden Blocks der Datei befindet. Die Mastercopy eines Blocks ist die erste Kopie, die irgendwo im kooperativen Cache gespeichert wurde. Die Liste mit den Hinweisen bekommt der Manager immer von dem Client, der zuletzt das Token für die bestimmte Datei bekommen hat, denn dieser Client hat wahrscheinlich die genauesten Informationen über den aktuellen Zustand des kooperativen Cache.
Schickt Client 1 eine Mastercopy an einen Client 2 weiter, weil Client 1 die Kopie löschen will (etwa aus Platzmangel), dann aktualisieren beide Clients ihre Hinweise dahingehend, daß jetzt Client 2 die Mastercopy besitzt.

4.4.2 „Block Lookup"-Mechanismus

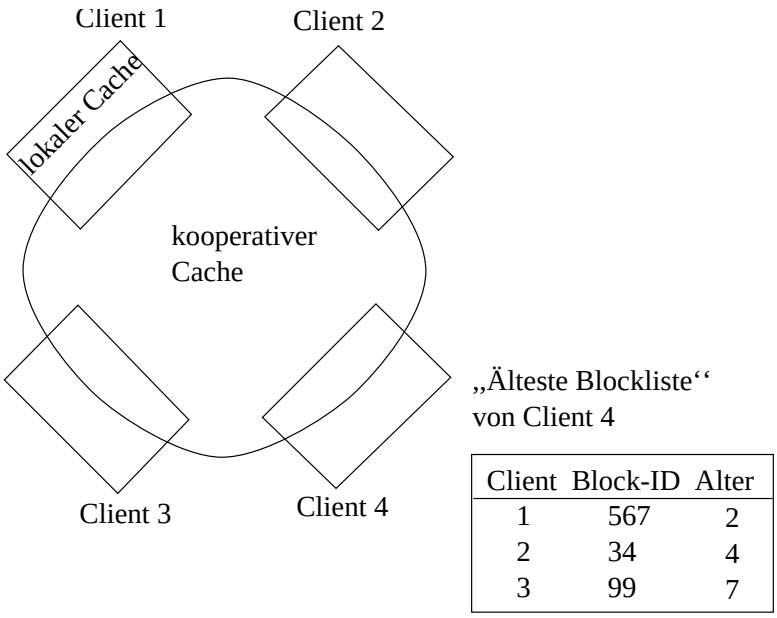
Die Hinweise, wo sich die Mastercopy eines Blocks befindet, müssen nicht immer richtig sein. In so einem Fall muß auf den Server zugegriffen werden, um den gesuchten Block zu erhalten. Da alle Schreibzugriffe dem Server mitgeteilt werden, besitzt dieser immer eine gültige Version aller Dateien. Der „Block Lookup"-Mechanismus sieht also folgendermaßen aus:

1. Findet ein Client den gewünschten Block nicht in seinem lokalen Cache, zieht er seine Hinweisliste zurate.
2. Ist ein Hinweis eingetragen, wo sich die Mastercopy des gesuchten Blocks finden läßt, dann wird die Anfrage an den entsprechenden Client gestellt. Anderenfalls wird die Anfrage direkt an den Server gesendet.
3. Der Client, der die Anfrage nach der Mastercopy erhält, schickt diese dem entsprechenden Client, falls er die Mastercopy tatsächlich besitzt. Im anderen Fall wird zu Schritt 2 gesprungen.

4.4.3 Ersetzungsstrategie

Entschließt sich ein Client, einen Block aus seinem lokalen Cache zu löschen, muß entschieden werden, ob der Block verworfen oder an einen anderen Client gesendet werden soll (Forwarding). Die Entscheidung hängt davon ab, ob der betreffende Block eine Mastercopy ist oder nicht, denn nur eine Mastercopy wird weitergeschickt. Damit wird vermieden, daß sich mehrere Kopien desselben Dokuments im kooperativen Cache befinden, was die Leistung des Caches reduzieren würde. Handelt es sich nicht um eine Mastercopy, dann wird der Block verworfen, d.h. er wird einfach aus dem Cache gelöscht. Anderenfalls muß bestimmt werden, an welchen Client der Block weitergeschickt werden soll. Falls der Cache dieses Clients voll ist, muß dieser Platz für den Block schaffen, d.h. er muß selbst auch wieder einen Block löschen.
Bisherige Algorithmen haben den Client, an den der Block geschickt wird, entweder zufällig bestimmt oder haben die Entscheidung dem zentralen Manager überlassen, was die Netzbelastung und die Antwortzeit erhöht. Dieser Algorithmus verwendet ein Verfahren, das *„best-guess"-Ersetzung* genannt wird. Jeder Client verwaltet eine Liste, in der der seiner Meinung nach älteste Block im Cache jedes anderen Client registriert ist. Abbildung 6 zeigt einige Clients, die einen kooperativen Cache bilden. Beispielhaft ist dabei die „älteste Blockliste" von „Client 4" abgebildet.
Ein Block wird nach dieser Strategie an den Client geschickt, der den insgesamt ältesten Block

in seinem Cache hat. Diese „ältesten Blocklisten" werden dadurch auf einem aktuellen Stand gehalten, daß zwei Clients ihre Listen miteinander abgleichen, wenn einer der beiden dem anderen einen Block schickt. Simulationen haben gezeigt, daß diese Strategie dem globalen



Client	Block-ID	Alter
1	567	2
2	34	4
3	99	7

Abbildung 6: Clients mit ihren „ältesten Blocklisten"

LRU ziemlich nahe kommt, trotz daß die „ältesten Blocklisten" nicht immer genau sind, da sie nicht global verwaltet werden. Um die Leistungseinbußen, die durch falsche Informationen aus der „ältesten Blockliste" entstehen, zu verringern, wendet man folgende Methode an:
Wenn ein Client eine Mastercopy eines Block weiterschicken will, so sucht er, wie oben schon erwähnt, seinen Ziel-Client aus seiner „ältesten Blockliste" aus. Wenn dieser Client aus Platzmangel in seinem Cache den ältesten Block zugunsten des gerade empfangenen ersetzen muß, prüft er anschließend, ob es laut *seiner* „ältesten Blockliste" einen Client gibt, der einen noch älteren Block in seinem Cache hat. Ist das der Fall, dann schließt er daraus, daß die „ältesten Blocklisten" offenbar falsche Informationen enthalten haben. In diesem Fall schickt er den gerade ersetzten Block quasi als Ausgleich für den Fehler zum Cache des Servers. Dieser Servercache wird als *„Discardcache"* bezeichnet und ist das „Auffangbecken" für Blöcke, die auf Grund falscher Informationen gelöscht wurden, obwohl sie nicht die global ältesten Blöcke waren. Der Discardcache selbst verwendet die LRU-Ersetzungsstrategie.

4.5 Leistungsanalyse

Wie schon weiter oben erwähnt, erreicht dieser Algorithmus mindestens die Blockzugriffszeit der bisherigen Algorithmen. Der große Vorteil wird aber erst deutlich, wenn man die Entlastung des Managers betrachtet. Abbildung 7 zeigt die Managerbelastung bei verschiedenen Algorithmen.

Man sieht, daß bei allen drei Algorithmen der Manageraufwand für die Aufrechterhaltung der Konsistenz ähnlich gering ist. Deutlich bessere Werte werden durch die Verwendung von Hinweisen aber bei der Blockersetzung und beim Block Lookup erreicht, da der Manager außer zur Tokenvergabe nur kontaktiert werden muß, wenn die Hinweise falsch waren, was offensichtlich nur sehr selten der Fall war (0,01%).
Verglichen mit N-Chance und GMS erreicht man durch die Verwendung des auf Hinweisen basierenden Algorithmus durchschnittlich eine Reduzierung der Managerlast um den Fak-

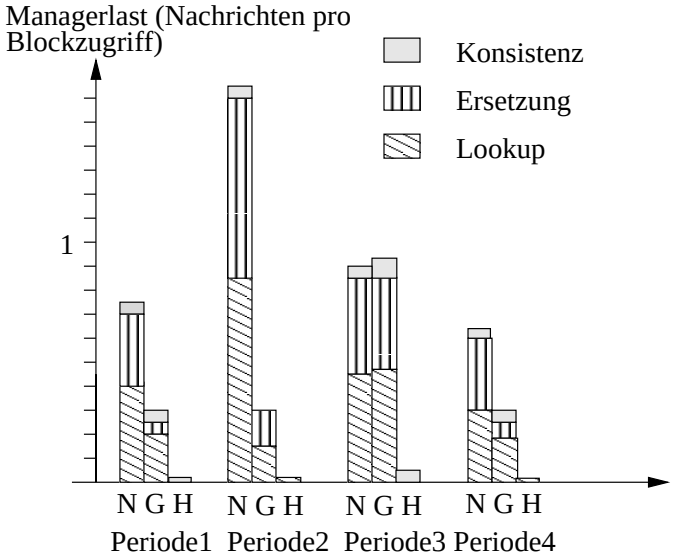


Abbildung 7: Managerbelastung. Vergleich des auf Hinweisen basierenden Algorithmus (H) mit den herkömmlichen Methoden N-chance (N) und GMS (G) über mehrere Perioden hinweg. Die Managerlast ist gemessen in der Anzahl der gesendeten und empfangenen Nachrichten pro Blockzugriff und aufgeteilt in die Ursachen der Kommunikation

tor 15, eine Reduzierung des „Block Lookup"-Verkehrs um den Faktor zwei drittel und eine Reduzierung des Verkehrs bei Ersetzungen um den Faktor 5.

5 Reduzierung der Server-Last und Antwortzeit durch server-seitige Spekulation

5.1 Einleitung

Server-seitige Spekulation bedeutet, daß ein Server nicht nur angeforderte Dokumente an den Client schickt, sondern noch weitere Dokumente, die der Client in naher Zukunft wahrscheinlich ohnehin anfordern würde. Die Entscheidung, welche Dokumente spekulativ geschickt werden, trifft der Server auf Grund von statistischen Informationen, die er für alle Dokumente, die er anbietet, verwaltet. Durch die Anwendung server-seitiger Spekulation lassen sich sowohl die Belastung des Servers als auch seine Antwortzeit erheblich verringern.

5.2 Zugriffsabhängigkeiten

Man sagt, zwischen zwei Dokumenten auf einem Server besteht eine Zugriffsabhängigkeit, wenn beide Dokumente innerhalb einer bestimmten Zeit vom selben Client angefordert werden. Es gibt verschiedene Stufen der Abhängigkeiten:

- *„embedding dependency"*: Ein Dokument 1, das in ein anderes Dokument 2 eingebettet ist, wird immer benötigt, sobald Dokument 2 angefordert wird, d.h. die Abhängigkeit beträgt hier 100%.
- *„traversal dependency"*: Ein Dokument 1 wird mit einer bestimmten Wahrscheinlichkeit kleiner 100% angefordert, falls Dokument 2 angefordert wird.

5.3 Systemmodell

Das Systemmodell ist relativ einfach. Erhält ein Server eine Anfrage für ein bestimmtes Dokument, dann schickt er außer dem gewünschten noch weitere Dokumente mit, die

- einen gewissen Schwellenwert für die Zugriffsabhängigkeiten überschreiten.
- einen gewissen Grenzwert für ihre Größe nicht überschreiten.

Die Zugriffsabhängigkeiten berechnet der Server in bestimmten Abständen aus seinen Log-Dateien.
Ferner wird vorausgesetzt, daß die Clients über einen Cache verfügen. Will ein Client dann auf ein Dokument zugreifen, das spekulativ gesendet wurde, so befindet sich dieses Dokument bereits in seinem Cache.

5.4 Simulationsergebnisse

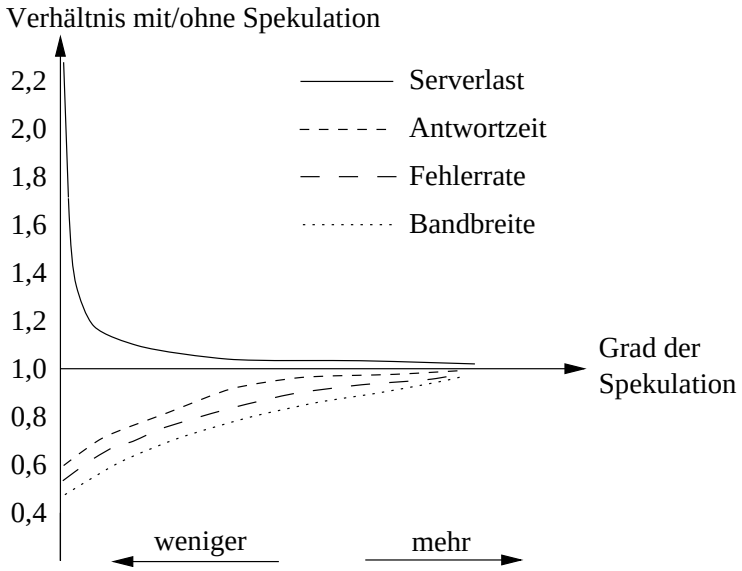


Abbildung 8: Ergebnisse für verschiedene Stufen der Spekulation.

Abbildung 8 zeigt die Reduzierung an Server-Last und Antwortzeit in Abhängigkeit des Spekulationsgrades. Außerdem ist die jeweilige Erhöhung des Bandbreitenbedarfs eingezeichnet. Man sieht, daß die Einsparungen im Verhältnis zur Zunahme des Bandbreitenverbrauchs am größten sind, wenn nur relativ vorsichtig spekuliert wird. Z.B. erreicht man eine Reduzierung der Server-Last um 30% und eine Verkürzung der Antwortzeit um 23% mit nur 5% zusätzlicher Bandbreite. Erhöht man den Grad der Spekulation soweit, daß man 10% zusätzliche Bandbreite benötigt, ist nur noch eine geringe Verbesserung auf 35% bzw. 27% möglich. Wenn also ein gewisser Punkt überschritten wird, dann zahlt sich die Spekulation kaum mehr aus.

Abbildung 9 verdeutlicht, daß auch der Grenzwert für die Größe der Dokumente, die spekulativ gesendet werden, einen großen Einfluß auf den Erfolg hat. Wie man sieht, liegt das Optimum für den Grenzwert bei relativ kleinen Dokumentgrößen. Das ist auch nachvollziehbar, denn wird einerseits ein Dokument spekulativ gesendet, ohne daß es später gebraucht wird, dann ist die dafür verschwendete Bandbreite für kleine Dokumente natürlich nicht so groß. Wurde andererseits richtig spekuliert, dann ist der Vorteil, den man durch kleine Dokumente erzielt, im Verhältnis größer. Denn der eingesparte Verschnitt, d.h. die Bandbreite für

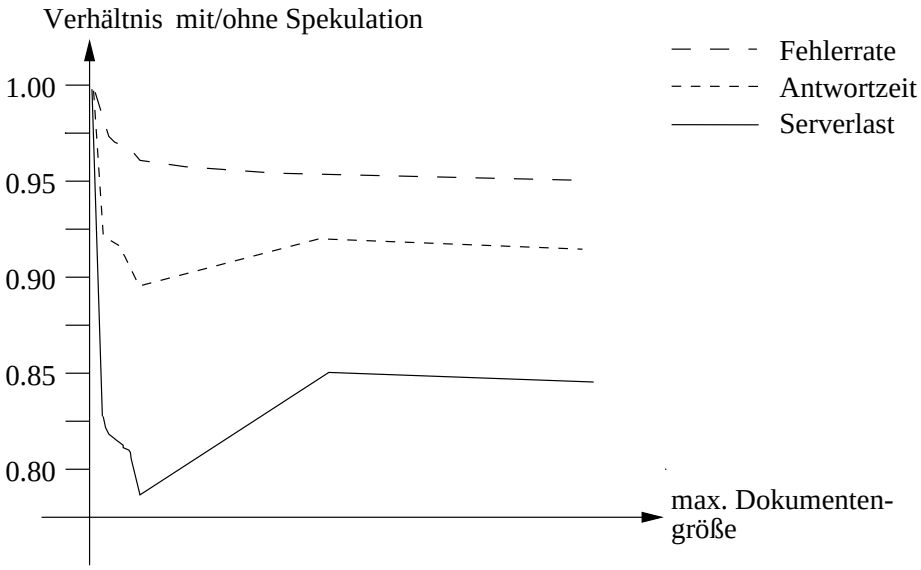


Abbildung 9: Abhängigkeit von der Dokumentengröße

die Anfrage an den Server und die Antwortzeit des Servers, ist im Verhältnis zur Dokumentengröße um so größer, je kleiner das Dokument ist.
Untersucht man die Zugriffsabhängigkeiten näher, stellt man fest, daß sie sich in der Zeit nur sehr langsam ändern. Das bedeutet, die Server müssen die Wahrscheinlichkeiten für die Zugriffsabhängigkeiten nur sehr selten neu berechnen, etwa alle 30 Tage.
Was die Größe des Clientcache betrifft, stellt man fest, daß bereits ein kleiner Cache ausreicht. Denn wenn ein spekulativ gesendetes Dokument im Cache nicht relativ schnell benötigt wird, dann ist die Wahrscheinlichkeit gering, daß es überhaupt gebraucht wird.

5.4.1 Verbesserungen

Eine weitere Leistungssteigerung ist möglich, wenn die Server berücksichtigen, welche Dokumente sie schon an welchen Client spekulativ gesendet haben. Denn bisher war es durchaus möglich, sogar wahrscheinlich, daß ein Server einem Client ein Dokument mehrmals schickt, obwohl dieses bereits im Cache des Client war. Um dem entgegenzuwirken, kann der Client bei einer Anfrage eine Liste mit den Dokumenten mitschicken, die er in seinem Cache hat. Dadurch wird das mehrfache Schicken derselben Dokumente an einen Client und die dadurch verursachte Bandbreitenverschwendung verhindert.

6 Vergleich

In diesem Bericht wurden vier neue Ansätze vorgestellt, die alle das Ziel haben, die verfügbare Bandbreite in verteilten Informationssystemen besser auszunutzen und die Antwortzeiten der Server zu verkürzen. Angesichts der rasant wachsenden Menge an verfügbaren Daten ist Skalierbarkeit eine wichtige Voraussetzung. Diese wird von allen vier Protokollen dadurch erfüllt, daß sie auf globale Kenntnis der Zustände verzichten und nur auf der Basis einfach und schnell erreichbarer lokaler Informationen arbeiten. Die Simulationen zeigen:

- Die Verwaltung globaler Informationen zahlt sich nicht aus.
- Ein zentraler Manager zur Steuerung ist unnötig.

- Lokale Kenntnisse über die Zustände in der nächsten Umgebung sind ausreichend, um eine bessere Ausnutzung der vorhandenen Bandbreite zu erreichen.

WebWave (Abschnitt 2) und die „nachfragegesteuerte Verbreitung von Dokumenten" (Abschnitt 3) versuchen, die Last auf viele Server zu verteilen. Das Problem bei WebWave ist, daß in die Router Paketfilter eingeschleust werden müssen. Außerdem ist die Voraussetzung unveränderlicher Dokumente sehr restriktiv. Will man das Protokoll auf Dokumente, die (möglichst selten) geändert werden dürfen, ausweiten, dann muß man zusätzlich den Aspekt der Konsistenz bzw. Kohärenz der Dokumente berücksichtigen, was das Protokoll natürlich entsprechend komplexer macht.

Vergleich			
	Konsistenz	Generalität	zusätlicher (Hardware-) Aufwand
WebWave [HeMi97]	stark, da nur unveränderliche Dokumente verteilt werden	gering, Voraussetzung: unveränderliche Dokumente	Paketfilter, Cacheserver
DDD-WWW [Best95a]	schwach, Konsistenzprotokolle: TTL, Alex	beschränkt: Dokumente sollten selten geändert werden	Serverproxy
kooperatives Caching [SaHa96]	schwach, Verwendung von Token	gut, keine besonderen Voraussetzungen	Teil des lokalen Caches ist Teil des kooperativen Caches
server-seitige Spekulation [Best95b]	schwach, keine Maßnahmen nötig, da Dokumente auf dem ursprünglichen Server bleiben	gut, keine besonderen Voraussetzungen	keiner

Tabelle 1: Vergleich der verschiedenen Ansätze

Beim „kooperativen Caching unter Verwendung von Hinweisen" (Abschnitt 4) wird die Netzentlastung dadurch erreicht, daß jeder Client selbst einen Teil der Cache-Verwaltung übernimmt und so auf die zentrale Steuerung verzichtet werden kann.

Durch „server-seitiges spekulatives Versenden von Dokumenten" (Abschnitt 5) erreicht man die besten Ergebnisse. Die Reduzierung der Server-Last und die Verkürzung der Antwortzeiten übertreffen sowohl die Ergebnisse, die durch kooperatives Caching als auch durch server-seitige Datenverteilung erreicht werden.

Weitere Leistungssteigerungen sind vermutlich durch die Kombination verschiedener Ansätze erreichbar. Am besten lassen sich wohl die Algorithmen „WebWave", „server-seitige Spekulation" und „nachfragegesteuerten Verbreitung von Dokumenten" mit dem Algorithmus für das „effektive kooperative Caching durch Verwendung von Hinweisen" kombinieren. Diese Kombination scheint sinnvoll zu sein, da sich die drei erstgenannten Algorithmen mit der Entlastung der Server befassen und der letztgenannte Algorithmus versucht, die Leistungsfähigkeit eines kooperativen Caches zu erhöhen.

Tabelle 1 zeigt einen Vergleich der vier verschiedenen Ansätze hinsichtlich Konsistenz, Generalität, d.h. allgemeiner Verwendbarkeit in verteilten Systemen, und zusätzlichem (Hardware-) Aufwand.

Deutlich wird bei allen vier Ansätzen, daß es sich in verteilten Informationssystemen nicht auszahlt, totale Kenntnis über den Zustand des Gesamtsystems zu anzustreben. Denn der zusätzliche Verbrauch an Bandbreite und der Verwaltungsaufwand, die nötig sind, stehen in keinem Verhältnis zu den minimalen Steigerungen bei einigen Bewertungskriterien. Die Gesamtleistung ist schließlich schlechter, als wenn man auf diesen „Allwissenheitsanspruch" verzichtet.

Literatur

[Best95a] Azer Bestavros. Demand-based document dissemination to reduce traffic and balance load in distributed information systems. In *Seventh IEEE Symposium on Parallel and Distributed Processing*, San Antonio, Texas, Oktober 1995.

[Best95b] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *CIKM '95: The Fourth ACM International Conference on Information and Knowledge Management*, November 1995.

[Best96] Azer Bestavros. Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time for Distributed Information Systems. In *ICDE '96: The 1996 International Conference on Data Engeneering*, New Orleans, Louisiana, März 1996.

[Cate92] V. Cate. Alex - a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop*, Ann Arbor, MI, Mai 1992.

[DWAP94] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson und David A. Patterson. Cooperative Caching: Using Remote Client Memory to improve File System Performance. In *Proceedings of the 1st Symposium on Operating System Design and Implementation*, November 1994, S. 267–280.

[FMPK+95] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin und Henry M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th Symposium on Operating System Principles*, Dezember 1995, S. 201–212.

[HeMi97] Abdelsalam Heddaya und Sulaiman Mirdad. WebWave: Globally Load Balanced Fully Distributed Caching of Hot Published Documents. In *International Conference of Distributed Computing Systems*, Baltimore, Maryland, Mai 1997. S. 160–168.

[SaHa96] Prasenjit Sarkar und John H. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Oktober 1996, S. 35–46.

Dynamisch verteilte Objekte

Matthias Bader

Kurzfassung

In Zukunft werden für Software zwei Leistungsmerkmale unabdingbar sein, um im Markt bestehen zu können: Verteilung (alle großen Firmen sind zur Zeit dabei, ihre großen, monolitischen Programme zu zerstückeln, um sie flexibler für Wartung und Updates zu machen und als verteilte Anwendungen auf verschiedenen Rechner laufen zu lassen) und Flexibilität (Komponenten sollten sich einfach austauschen oder verschieben lassen). Es existierten zwar einige propritäre Systeme, aber offene Standarts (wie HTML oder CORBA) sind nicht in Sicht. Wie müssten sie aussehen? Dieser Artikel stellt einige Ansätze vor.

1 Einleitung

Die Softwaretechnik hat seit ihrer Entstehung bereits zwei Paradigmenwechsel erlebt. Am Anfang war der Mainframe, bedienbar über "dumme" Terminals, die einzige mögliche Existenzform für große Systeme. Mitte der 80er Jahre vollzog sich, bedingt durch die Verfügbarkeit billiger PCs, ein Wandel hin zum Client-Server-Modell. In den letzten Jahren ist wiederum ein Wechsel zur Middelware zu beobachten, d.h. die scharfen Grenzen zwischen Klient und Server verschwimmen. Man kann damit die ganze Bandbreite zwischen "Thin-" und "Fatclient"-Architekturen abdecken und für jeden Anwendungsfall die optimale Konfiguration anbieten.

Es ist längst abzusehen, worin diese Entwicklung ihren Fortgang haben wird. In den bestehenden verteilten Systemen ist das Austauschen oder Hinzufügen einzelner Komponenten aufwendig und schwierig, zur Laufzeit sogar unmöglich, da diese an einen bestimmten Ort (Rechner, Port) gebunden und meist als zu grobkörnige, monolitische Programmfragmente realisiert sind [LaVe97].

Diese Nachteile zu entfernen, ist das Ziel einiger neuer Forschungsarbeiten. Deren Ansätze sind unterschiedlich: Einige unterstützen den Entwickler nur bei der Erstellung interagierender Komponenten (Abschnitte 4.1 und 4.2), andere verhelfen den verbundenen Objekten zu Dynamik und Flexibilität zur Laufzeit (Abschnitte 5.1 und 5.2).

Das in Abschnitt 3 beschriebene Konzept setzt am dienstgebenden Objekt (oder Komponente) selbst an: Es wird zerteilt und dynamisch verteilt.

Anfangs wird auf die Techniken laufzeitabhängiger Verknüpfungen von funktionalen Bibliotheken mit Anwendungen verwiesen. Auch hier ergeben sich nämlich seit geraumer Zeit interesante Alternativen zu herkömmlichen Verfahren.

Nach kurzer Betrachtung sind drei Kriterien erkennbar, nach denen sich die aufgeführten Konzepte sinnvoll einordnen lassen. Je nach Schwerpunkt differenziert man nach

- *Dynamik der räumlichen Verteilung*, also nach dem Maß, nach dem es ein Modell erlaubt, ganze Objekte ohne großen Aufwand von (Netzwerk-)Ort zu Ort zu verschieben, ohne daß die umschließende Applikation gestört wird.

- *Dynamik der Struktur*. Ein Objekt, das beispielsweise erst auf Anforderung "zusammengebaut" wird, besitzt in dieser Kategorie eine hohe strukturelle Dynamik, fest verdrahtete Strukturen eine niedrige strukturelle Dynamik.

- Wichtig ist nicht zuletzt der Aspekt des *Kommunikationsbedarf* . Er beschreibt vor allem die Eignung für LANs oder WANs, lokales oder mobiles Umfeld (Ein hoher Kommunikationsaufwand ist im mobilen Einsatz ungeeignet, weil Totzeiten zu lang werden und die Verbindung sogar ganz unterbrochen werden kann).

Weitere Kriterien sind denkbar. Aber allein der Versuch, die realisierte Architektureinzuordnen, schlüge mit mehreren Dimensionen zu Buche, zumal da einige hier vorgestellte Verfahren dem Entwickler die Wahl einer geeigneten Vernetzung seiner Komponenten überlassen. So sollen die oben genannten Kriterien die einzigen bleiben.

2 Werkzeuge für dynamische Verteilung

Dieses Kapitel stellt noch kein verteiltes Objektmodell vor. Vielmehr sollen einige Möglichkeiten für dynamisches Binden Erwähnung finden.

Das Binden (engl.: *linking*) hatte ursprünglich zum Ziel, eine Möglichkeit zu bieten, um die großen, monolitischen Programme der Mainframes zu zerteilen und immer wieder verwendeten Code in Bibliotheken systemweit zur Verfügung zu stellen. Aber man erkannte bald, daß damit auch dynamische Effekte erreicht werden können. Hierzu muß ein Compiler für einen Bibliotheksfunktionsaufruf einen Verweiß erzeugen, der in einer separaten Verweistabelle aufgelistet ist. Es ist nun Aufgabe des (dynamischen) Linkers, die Aufrufe aufzulösen. Man erhält damit Unabhängigkeit von der Lokalität und der Implementierung der Module.

Aber zur Laufzeit muß jeder Aufruf dereferenziert werden, außerdem kann die Verweis-tabelle nicht auf dem neuesten Stand gehalten werden, weil z.B. zwischenzeitlich das Modul verschoben wurde. Das alles ist also sehr zeitaufwendig und fehlerträchtig.

Eine verbesserte Methode besteht darin, den Linker zur Laufzeit den Code modifizieren zu lassen: die Verweise bleiben dabei zunächst unaufgelößt. Erst wenn man bei der Abarbeitung auf eine Bibliotheksroutine trifft, setzt der Linker die genaue Adresse für den Sprung ein.

Ein vielversprechender, neuer Ansatz setzt auf *Codeerzeugung zur Laufzeit*. Also eine Art Just-in-Time-Compiler. Die von M. Franz beschriebene Technik [Fran97] teilt den Compiler in zwei Teile auf: der erste Teil erzeugt aus dem Quell- eine Art Bytecode, der zweite Teil produziert zusammen mit dem Linker erst zur Laufzeit Maschinencode.

Die Vorteile sind einleuchtend: der Bytecode ist im Schnitt 30 % kompakter als Maschinencode, er ist nicht hardwareabhängig, Optimierung kann über Modulgrenzen hinweg erfolgen und schließlich erhält man ja faktisch einen Applikationsmonoliten (was für die Ausführung immer besser ist), ohne seine Nachteile bei der Entwicklung in Kauf nehmen zu müssen.

Bislang besaßen Kritiker dieser Methode (die es schon länger gibt) immer das Argument des Performanzverlustes gegenüber dem fertigübersetztem Programm. Doch das ist mittlerweile entkräftet. Hintergrund ist die im Vergleich zu der I/O-Hardware sehr viel schneller wachsende Leistungsfähigkeit heutiger Prozessoren. Während die alte Methode des Linkens (durch die viele externen Aufrufe) vor allem Bus und Speicher beanspruchte, ist beim neuen Konzept hauptsächlich die Rechenleistung gefragt.

Nur der Vollständigkeit wegen sei hier noch die Klasse der voll laufzeit-compilierten Ausführungsstrategien erwähnt. Der größte Nachteil gegenüber dem obigen Verfahren ist die Tatsache, daß das Parsen, eine Aufgabe, die der erste Teil des Compilers erfüllt, statisch aber aufwendig ist. Auch diesen Part jedesmal bei einem Aufruf durchzuführen, bedeutet einen überflüssigen Arbeitsaufwand. Außerdem ergibt sich ein rechtliches Problem des Urheberschutzes, da der (lesbare) Quellcode einsehbar ist.

3 Verteilte, dynamisch konstruierbare (Server-)Objekte

Verteilte Systeme bestehen aus zwei Ebenen, den Objekten und deren Vernetzung. Dynamik erreicht man folglich, indem man die Objekte oder/und die Vernetzung flexibel gestaltet. Gründer et al. beschreiben in ihrem Ansatz [GGKB98] den ersten Weg über die Flexibilisierung der Objekte.

3.1 Theoretische Beschreibung und Implementation

Gründer, Geihs, Knape und Baumert zerlegen Objekte in drei grundsätzliche Teile:

- **Identität** : Bezeichnet eine Instanz, die z.B. mit *new()* erzeugt wird.
- **Zustand** bezeichnet im allgemeinen die Daten(felder) einer Instanz.
- **Verhalten** : im weitesten Sinne alle Objektfunktionen (Methoden).
  Das Verhalten ist immer vom aktuellen Zustand abhängig.

Die heutige OOP trennt Identität und Zustand nicht, gleichwohl aber das Verhalten. Aber auch diese Teilung relativiert sich, wenn man bedenkt, daß alles durch einen Compiler verarbeitet werden muß. Von Dynamik kann keine Rede sein: Eine Instanz residiert auf **einem** Rechner, in **einem** Addressraum, mit Code und Daten.

Die Autoren gehen hier einen neuen Weg. Indem sie ein Objekt in drei Teile aufspalten und diese nur noch mittels Referenzen aufeinander zugreifen lassen, erreichen sie eine Flexibilität, die die einleitend genannten Gegensätze von Modell und Implementation überwinden hilft.

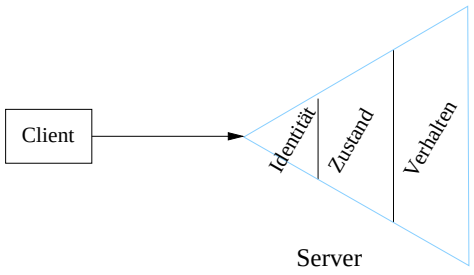


Abbildung 1: Ein Client-Server-Modell vor . . .

Abbildung 2 zeigt, daß

- ein Klient eine Referenz auf die Identität des Server-Objektes hält.
- der Zustandspart zu *Chassis* umbenannt wird und die Identität zu *Metachassis*.

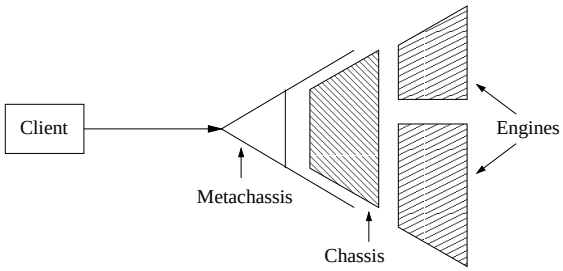


Abbildung 2: . . . und nach der logischen Trennung

- die Autoren die Objektmethoden als *Engines* bezeichnen.
  Dabei fällt auf, daß diese nun in mehrere Teile zerfallen sind, ebenso wie das Chassis, wie es auch Abbildung 3 zeigt.

Infolge der Trennung der Engines vom übrigen Objekt ist es nun einfach, das Gesamtverhalten in Einzelteile zu zerlegen und diese sogar auf unterschiedliche Rechner zu verteilen. So können Persistenzaufgaben auf Datenbankserver und GUI-Berechnungen auf spezielle Grafik-Workstations gelegt werden. Es ist auch zu beachten, daß unterschiedliche Versionen der gleichen Methode auf der gleichen oder einer anderen Maschine residieren können. Beispielsweise existiert die Standardversion einer Bildausgabefunktion auf dem Arbeitsplatzrechner, ein Delux-Exemplar (mit rendering) auf besagter Grafik-Workstation. Der Klient entscheidet im Zweifelsfall (oder entgegen einer Default-Einstellung), was benutzt wird. Man kann sich leicht vorstellen, daß mit einem analogen Verfahren auch echte parallele Verarbeitung ohne Multiprozessorsysteme möglich ist.

Es leuchtet ein, daß Fragen nach Konsistenz und Synchronisation hier in den Methoden selbst behandelt und gelöst werden müssen: Paradebeispiel wäre eine Speicherfunktion, die von vielen Objekten gleichzeitig um Speicherdienste gebeten wird; sie darf nicht mehrfach ausgeführt werden, oder muß zumindest selbst für eine Synchronisation der Zugriffe sorgen.

Oft tritt bei unterschiedlichen Versionen das Problem auf, daß sie auch verschiedene bzw. zusätzliche Zustandsvariablen erfordern: die Grafik-Workstation, um dieses Biespiel weiterzuverwenden, verlangt vieleicht eine doppelte Genauigkeit der Koordinaten gegenüber der Standardvariante. Es ist somit sinnvoll, das Zerteilungsprinzip der Engines auch auf die Chassis anzuwenden. Denkbar ist ein allgemeine Teil des Chassis, den jede Implementiertung des Grafikalgorithmus verwenden kann, und ein spezifischer Teil, der je nach Rechner die einfache oder die doppeltgenaue Variante beinhaltet.

Ohne die Schnittstelle des Klienten zum Objekt zu berühren, kann eine Änderung zur Laufzeit nur für die Cassis und die Engines erfolgen. Das heißt, das Metachassis bleibt unverändert und für den Klient läuft alles selbst zur Laufzeit transparent ab.

Abbildung 3 erläutert eine Weiterentwicklung des Objektes gemäß dem vorgestellten Modell.

Um das beschriebene Modell zu verwirklichen, wurde eine CORBA-Implementierung eingesetzt. Sie erhielt eine Erweiterung des Interface-Repository und des Implementation-Repository. Der IDL-Compiler wurde stark erweitert, damit die neue Betrachtungsweise auch modelliert werden kann. Der Compiler erstell jetzt neben den Stubs und den Skeletons zusätzlich die Metachassies und Chassies der definierten Objekte, sowie die “Skeletons” der Engines. Die Aufgabe der Entwicklers ist, wie bisher, nur die Implementierung der Methoden des Objektes.

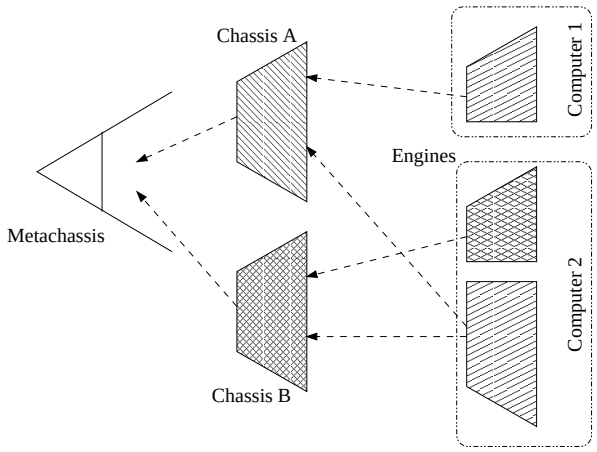


Abbildung 3: Verteiltes Objekt

Das vorgestellte dynamische Modell verbirgt die zugrundeliegende Komplexität vor dem Programmierer, nicht aber seine Entscheidungen, die die Gesamtstruktur betreffen. Diese sind vor allem

- Zugriffsberechtigungen,
- Position der einzelnen Komponenten und natürlich
- die Frage, welches Chassis mit welcher Engine arbeiten kann und soll.

Wichtig ist in diesem Zusammenhang nochmals anzumerken, daß eine einmal beschlossene Architektur nicht eingefroren wird. Selbst und gerade zur Laufzeit ist eine komplette Neukonfiguration möglich.

3.2 Vererbung und Wiederverwendung dyn. Objekte

Die Objektorientierung wurde deshalb so erfolgreich, weil sie Vererbung, Polymorphie und einen einfachen Mechanismus zur Wiederverwendung bietet. All diese Eigenschaften sind in verteilten Systemen nur mit Mühe zu realisieren, denn Vererbung und polymorphe Ersetzbarkeit von Objektklassen durch ihre abgeleiteten Kinderklassen benötigen einheitliche Signaturen, Verhalten und Implementierung. Während Vererbung bei Objektsignaturen (Interfaces) noch gut realisierbar ist (in IDL möglich), ist dasselbe beim Letztgenannten ungleich schwieriger zu erreichen. Wie soll ein Objekt, das in Java erstellt wird, eine ererbte Funktion der Elternklasse (C++), aufrufen können? Die Mechanismen, die bis dato möglich sind, nämlich Aggregation und Delegation, sind kein vollwertiger Ersatz für Vererbung. Es fehlen vor allem Automation und konsistente Namensräume. [GrGe96]

Das verteilte Objektmodell von Gründer und Geihs kann hier eine Lösung darstellen.

Folgende Situation ist denkbar: eine Klasse C ist Mehrfacherbe von A und B. Das kann ohne weiteres in IDL modelliert werden. Und wie in einer "normalen" OO-Applikation implementiert der Programmierer ebenfalls nur die Methoden, die eine zusätzliche Funktionalität für die Klasse C bereitstellen, bzw. überschreibt bestehende und vererbte. Der modifizierte Compiler kreiert daraufhin ein neues Metachassis. Dieses verweist wiederum auf ein neues Chassis, in das die Chassis der Klassen A und B und noch ein weiteres Fragment, das die Felder und Zustände für die neuen Methoden von C enthällt, eingebunden werden. Das zusätzliche

Fragment besitzt zudem Referenzen auf die neuen Methoden, genauso wie die Chassis von A und B die Referenzen auf die alten vererbten Methoden aus A und B beinhalten.

Die strenge Trennung von Identität, Zustand und Verhalten garantiert, daß man das gewünschte Objekt erhält.

4 Statische, fragmentierte Objekte

Die beiden folgenden Konzepte erheben nicht den Anspruch, dynamisch zur Laufzeit zu sein. Sie zielen nur darauf ab, den Vorgang des Verteilens so weit wie möglich zu vereinfachen und zu automatisieren.

4.1 Entwickler-gesteuerte Verteilung

Das Modell der Objektfragmentierung von Makpangou, Gourhant, Le Narzul und Shapiro [MGNS94] will eine einfache Erstellung eines verteilten, mehrfach genutzten Objekts erreichen. Solch ein Objekt teilen die Autoren zuerst einmal in Fragmente auf. Das sind selbst wiederum Objekte mit öffentlicher Schnittstelle und privatem Implementationsteil. Zusätzlich beinhalten sie aber noch einen Kommunikationsabschnit, der die Verbindung zu den anderen Fragmenten des verteilten Gesamtobjekts sicherstellt. Diese Einteilung geht allerdings weiter, als es den Anschein hat, was sich in der dafür extra entwickelten Sprache FOG (Fragmented Objekt Generator) niederschlägt: Bei der Definition eines Fragments steht neben den "normalen" *privat*, *protected* und *public* Schlüsselwörtern auch noch *group* zur Verfügung. In dem dadurch gebildeten Abschnitt finden die Funktionen und die Daten, die andere Fragmente bereitstellen, Platz.

Um die Verständigung möglichst einfach zu halten, kann ein Fragment auf einen ganzen Satz von vorgefertigten Kommunikationsobjekten zurückgreifen, wobei das Spektrum von der Punkt-zu-Punkt-Verbindung bis zum gepufferten Mehrfachkanal reicht (Abbildung 4).

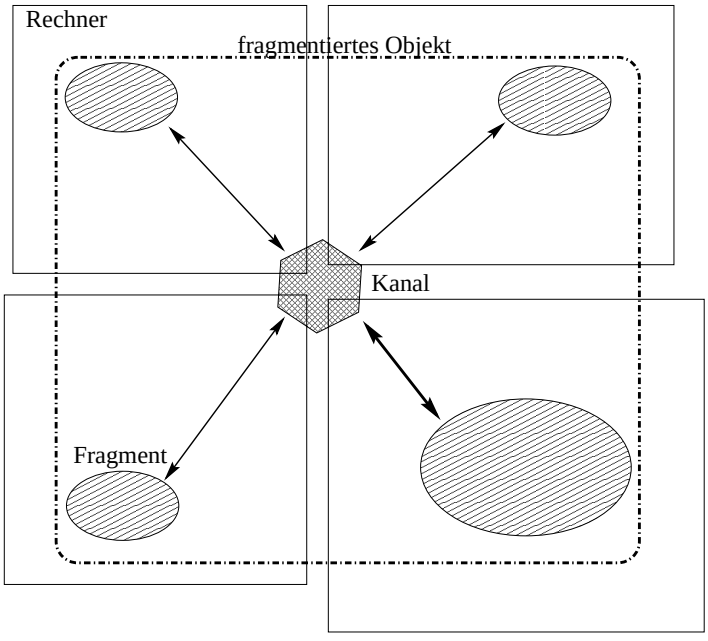


Abbildung 4: Fragmentiertes Objekt

Der spezielle Compiler leistet die Hauptarbeit: er erstellt für die Funktionen des *group*-Abschnitts die RPC-Stubs, die die Aufrufe über das Netz weiterleiten, so daß der Entwickler nur eine leicht veränderte Syntax verwenden muß, um die Methoden verwenden zu können.

Zu beachten ist ferner, daß ein Fragment nicht an ein bestimmtes Objekt gebunden ist. Es muß erst zu diesem dazugebunden werden. Es bestehen dabei drei Möglichkeiten:

- lokales Binden, d.h. ein auf dem lokalen Rechner installierter Proxy stellt die Verbindung her,
- systemweiter, zentraler Verbindungsdienst und
- ein objektspezifischer Binder.

Mit diesem Ansatz ist die Gesamtstruktur des verteilten Objekts nicht festgelegt. Die klassische Client-Server-Strategie kann genauso modelliert werden, wie ein System ganz ohne Server, wo alle Daten und die gesamte Rechenleistung in den gleichwertigen Partnermaschinen residieren.

Die freie Wahl des Ortes der Fragmente und ihre Beziehung zueinander stellt sich aber oft als nachteilig heraus. In der Tat kann man durch eine unglückliche Plazierung einzelner Fragmente auf den falschen Rechnern ein wohlüberlegtes System ineffizient machen. Niemand wird den Entwickler davon abhalten können, zwei Komponenten, die viel miteinander kommunizieren, auf Maschinen zu verteilen, die nur über ein stark frequentiertes öffentliches WAN verbunden sind. Das gesamte Konstrukt wäre dann auf einen Bruchteil seiner Leistungsfähigkeit herabgesetzt.

4.2 Automatische Verteilung

Purao et all. ist es gelungen, die oben genannten Fehler dadurch zu vermeiden, daß sie einen Algorithmus entworfen haben, der automatisch die Programmodule zu Pakten zusammenstellt und diese dann auf die am besten dafür geeigneten Maschinen verteilt [PuJN98]. Um diese Aufgabe sinnvoll zu erledigen, benötigt der Algorithmus mindestens diese Parameter:

- die *Netztopologie* mitsamt der Leistungsfähigkeit der einzelnen Verbindungen und insbesondere, ob es sich um ein
- *offenes Netz* oder ein *schnelles LAN* handelt, sowie
- welche Rechner welche Besonderheiten haben, also wo sind die benötigten Datenbanken, wo ist der Endverbraucher u.s.w.

Dabei geht der Algorithmus in zwei Phasen vor: zuerst werden die Module gruppiert, die viel miteinander kommunizieren. Der Verkehr soll, wenn er über langsame öffentliche Netze laufen muß, minimiert werden. Die zweite Phase ist nur bei Einsatz schneller lokaler Netze sinnvoll: hier zerlegt man die Module wieder in kleinere Teile, die dann auf die Rechner verteilt werden. Denn jetzt stehen Kriterien im Vordergrund, die die Leistungsfähigkeit und speziellen Einsatzgebiete der existierenden Rechner berücksichtigen.

4.2.1 1. Phase

Wenn man sich frägt, wo man Applikationen am besten auftrennt,kommt man zu dem Schluß, daß wieder vom streng gekapselten Objekt Abstand genommen werden muß, denn OO-Software basiert auf Vererbung und das bedeutet, der meiste Verkehr findet zwischen den Eltern- und Kind-Objekten statt. Um das Kommunikationsaufkommen zu minimieren, müssen schon die Implementierungen aller Elternklassen in die Pakete übernommen werden.

Das ist auch die Vorgehensweise des Algorithmus.Allerdings bestimmt er nur die Komponenten der Elternklassen, die von der Kindklasse (direkt oder indirekt) benutzt werden, extrahiert sie und fügt sie dem neu entstehenden Paket zu.

Es bleibt zu bedenken, daß bei Paketbildung immer zwischen Größe und Verkehrsaufkommen abgewogen werden muß. Das kann der Algorithmus nicht alleine tun. Er kann nicht ohne weiteres erkennen, ob eine Komponente unbedingt ins Paket gehört, oder ob es genügt, diese per Fernzugriff zu referenzieren. Der Programmierer muß sie folglich zuerst mit einem "Prioritätsattribut" versehen.

4.2.2 2. Phase

Die im ersten Teil erzeugten Pakete würden die Vorteile schneller LANs nicht ausnützen. Deshalb zerlegt man sie wieder. Je nach dem, welche Maschinen vorhanden sind, werden für sie "Päckchen" gefertigt.

Denkbar ist z.B. die Datenbankkontrolle in einem "Päckchen" zu sammeln, ebenso die GUI-Module u.s.w. Soweit möglich müssen hier vom Programmierer auch Aspekte der Parallelität und der Konsistenz als Argumente eingebracht werden, damit diese mitberücksichtigt werden können.

Nachdem das alles entschieden ist, muß jedes Teilstück auch zu der Maschine gelangen, für die es bestimmt ist: das DB-Modul zum DB-Server und die GUI beispielsweise auf die SGI-Workstation.

Um auch diesen Arbeitsschritt automatisieren zu können, ist eine genaue Kenntnis von vorhandenen Rechnern, deren Einsatzmöglichkeiten und -Stärken notwendig. Aufgabe des Entwicklers ist es also, mittels zusätzlicher Attribute diese Zusammenhänge aufzuzeigen, damit der Algorithmus seinerseits korrekt arbeiten kann.

5 Dynamische, fragmentierte Objekte

Es liegt nahe, das Konzept der dynamischen Verteilung und das der Fragmentierung zu verbinden und so die Vorteile beider Welten zu nutzen.

5.1 Klient-gesteuerte Fragmentierung

Am Karlsruher Institut für Telematik wurde das Modell der *skalierbaren Dienstobjekte* [GrDR98] zu diesem Zweck entworfen. Grundlage bildet bei diesem Ansatz ein system- oder zumindest domainweiter Vermittlungsdienst, wie ihn viele gebräuchliche Middelwareprodukte enthalten, hier im speziellen der CORBA-Trading-Dienst. Da dieser aber nur unter starkem Vorbehalt als dynamisch zu bezeichnen ist, wurde er um einige grundlegende Funktionen erweitert.

Doch zuerst ein Blick auf die zugrundeliegende Theorie: Wie bei Gründer et al. wird der Server, oder genauer Fragmente von ihm, in die bekannten Teile Instanz, Zustand und Verhalten zerlegt. Grosses Weiterentwicklung besteht nun darin, den Dienstnehmer entscheiden zu lassen, ob er, wie bisher, nur mittels Referenzen darauf zugreifen will, oder ob er Teile der Implementierung und sogar die Zustandsvariablen bei sich lokal installieren will.

Hierfür können folgende Gründe bestehen:

Insbesondere im Bereich weiträumig verteilter Anwendungen oder im Mobilrechnerumfeld kann eine effiziente Kommunikation zwischen den Objekten in Folge von Verbindungsabbrüchen, mangelnder Kommunikationskapazität und Funkschatten nicht immer sichergestellt werden. Ein weiterer Nachteil der ausschließlichen Vermittlung von Objektreferenzen liegt in den langen Laufzeiten, die mit entfernten Objektaufrufen verbunden sind. Die lokale Objektinstantiierung ermöglicht in diesen Fällen ein (eingeschränktes) Weiterarbeiten der Anwendung. Es lassen sich so auch andere mobile Code-Paradigmen wie Remote Evaluation oder Code on Demand flexibel nutzen. Gleichzeitig ist durch die erweiterte Vermittlung immer gewährleistet, daß die aktuelle Version des Dienstes eingesetzt wird, selbst wenn dieser sich fast vollständig beim Klienten befindet. Schließlich können die Dienste über die Grenzen eines Traders hinweg angeboten werden und bleiben nicht, wie die dynamischen Objekte aus Abschnitt 2, an eine CORBA-Domäne gebunden.

An dieser Stelle muß auch das Problem der Synchronisation genannt werden: es liegt nicht im Aufgabenbereich der Traders, die Konsistenz das Objektzustandes zu gewährleisten. Dies muß durch den Server selbst geschehen.

Auch bei der Implementierung zeigt sich ein weiterer Nachteil; der vermittelte Code sollte ja auf jeder beliebigen Klientenmaschine ausführbar sein. Bereits compilierter Code ist dazu nur mit viel Mühe zu bewegen, deshalb ist der Prototyp des Traders und der Serverobjekte in Java erstellt worden. Hierbei sind die mobilen Programmstücke in JAR-Archive gepackt, was die Standardisierung erleichtert.

Eine Client-Anwendung hat dann die genannten drei Möglichkeiten, einen Dienstgeber zu instanziieren und auf diesem eine gewünschte Methode aufzurufen:

- Durch den konventionellen Methodenaufruf über CORBA (Abbildung 5);

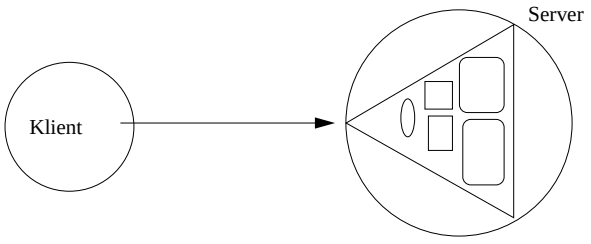


Abbildung 5: Konventioneller Methodenaufruf

- Anstelle eines Proxy-(Stub-)Objekt wird durch eine Hilfsklasse eine lokale Instanz (oder Teile) des Dienstobjekts erzeugt. Das angeforderte Codefragment wird heruntergeladen und an einem angegebenen Ort auf der Platte abgelegt, wo es gestartet werden kann und bei Bedarf mit den anderen (nicht lokal vorhandenen) Teilen seines Dienstobjektes Kontakt aufnimmt. (Abbildung 6)

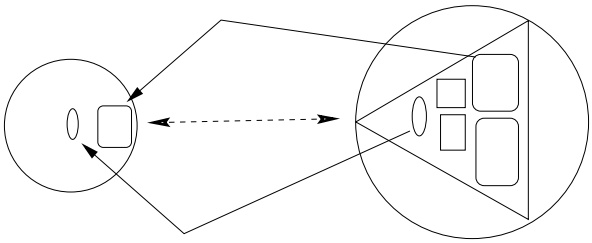


Abbildung 6: Kopieren von Code-Fragmenten

- Der dritte Fall ergibt sich durch Verwendung eines zusätzlichen Arguments bei der Dienstanfrage an den Trader: der Aufruf der Erzeugerfunktion stellt wie im zweiten Fall eine lokale Instanz des Objektes zur Verfügung und belegt unter Verwendung der Java-Serialization-Funktionalität die erzeugte Instanz mit dem zuvor von der Server-Anwendung exportierten Dienstobjektzustand.(Abbildung7)

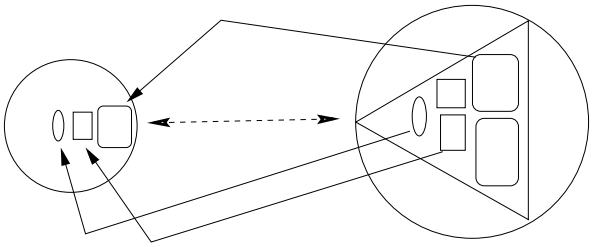


Abbildung 7: Kopieren von Daten und Code

5.2 Server-gesteuerte Fragmentierung

Globe von Tannenbaum et al. [STKS98] ist eine Mischung aus den Techniken in Abschnitt 5.1 und vor allem Abschnitt 4.1, geht aber bezüglich räumlicher und struktureller Dynamik noch weiter.

Der Hauptaugenmerk wurde auf Skalierbarkeit gelegt, da es für den Einsatz im Internet vorgesehen ist. Den Autoren fiel auf, daß z.B. persönliche WWW-Seiten letztlich mit der gleichen Technik publiziert werden wie große Web-Verzeichnisdienste. Es wird nicht zwischen zeitkritischen, großen Datenbeständen und selten veränderten Informationen unterschieden.

Auch bei Globe handelt es sich um ein verteiltes dynamisches Objektmodell, daß wie das Modell der fragmentierten Objekte 4.1 dem Klienten eine einfache Schnittstelle bietet und nicht an eine bestimmte Verteilungsstrategie gebunden ist. Im Unterschied zu jenem läßt dieses Modell aber obendrein freie Wahl bei den Caching-, Replikations- und Konsistenzkonzepten.

Drei übliche Caching-Methoden stehen zur Verfügung (Abbildung 8):

- alle Zugriffe erfolgen auf eine zentrale Kopie (a),
- es werden einige feste Kopien bestimmt(b) oder
- jeder Benutzer erhält eine lokale Version des Objektes (c).

Mit steigender Anzahl der Kopien erhöht sich natürlich auch der Aufwand, der nötig ist, um Konsistenz zu gewährleisten.

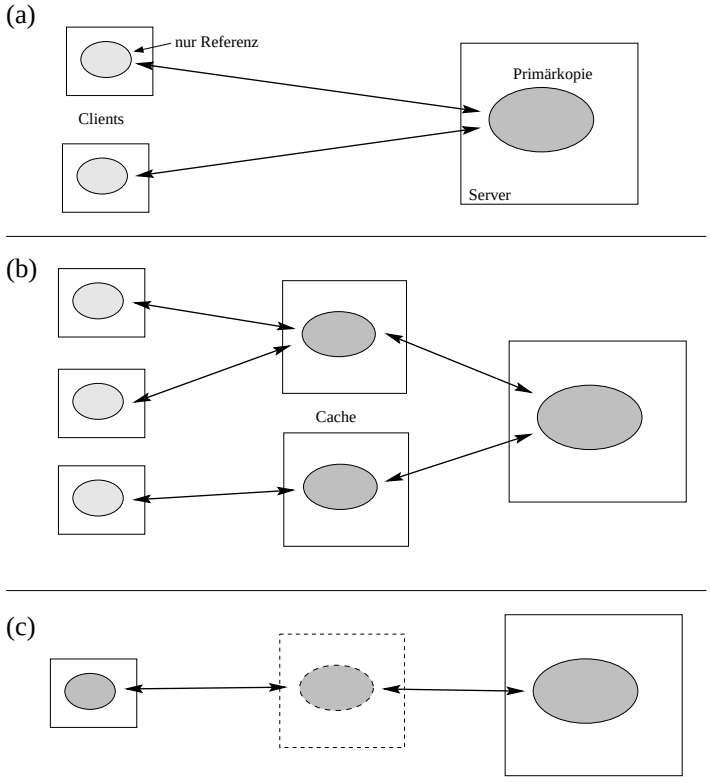


Abbildung 8: Caching-Methoden in GLOBE

Nur bedingt unabhängig vom Zwischenspeichern ist die Replikationsstrategie wählbar, denn erst die entsprechende Caching-Methode erlaubt die Entscheidung, ob ein Schreibzugriff nur auf der Primärkopie erfolgen darf oder die einzelnen Kopien sich synchronisieren sollen. Auch an dieser Stelle ist Globe flexibel: Die obige Entscheidung darf ein Klient sogar selbst treffen, wenn es ihm erlaubt ist. Er muß nur statt dem Kontaktpunkt des Proxy-Servers den der Primärkopie wählen. Steht diese Funktionalität zur Verfügung, steigt allerdings der logistische Aufwand zur Erhaltung der Konsistenz auf der Server-Seite rapide an.

Kernstück des Ganzen ist wie in Abschnitt 4.1 ein Dokument, ein GlobeDoc, das starke Ähnlichkeit zu HTML besitzt. Folglich kann es auf mehreren verteilten Web-Seiten liegen. Die dynamischen Teile sind als Hyperlinks auf ausführbaren Code (Applet/Servlet) realisiert. Dieser steht in sogenannten Klassenarchiven, die zur Laufzeit je nach Bedarf gruppiert werden können, um eine entsprechende Strategie zu implementieren. Das Interface eines GlobeDocs wird durch eine Globe-IDE verfügbar gemacht (Abbildung 9). Ein Benutzer wählt hierzu mit einem Browser die URL eines Kontaktpunktes des GlobeDoc oder eines allgemeinen Namingservice an, dabei wird ein Applet heruntergeladen, das das lokale Objekt darstellt. Es lädt anhand einer Objekt-ID aus einem Implementationrepository die nötigen Klassenarchive und initialisiert diese (ähnlich wie bei der Technik in 5.1). Was der Klient nicht sieht: die angeforderte Adresse muß nicht tatsächlich der Dienstgeber sein, es kann auch nur ein Cache-Manager sein, der die Operationen des Benutzers an eine in der Nähe liegende Kopie weiterleitet, oder gleich eine lokale Kopie zurückschickt.

Da dieser Ansatz von vornherein für das offene Internet konzipiert ist, wird sogar die Schaffung eines neuen Protokolls für Webbrowser erwogen, um nicht (wie bislang) an Java allein gebunden zu sein. (Ein GlobeDoc wird dann vermutlich mit einer Speziellen URL, wie z.B. `globe://informatik.ira.uka.de/seminar` referenziert).

Noch sind Aspekte der Sicherheit nicht gelöst. Daran wird aber derzeit geforscht.

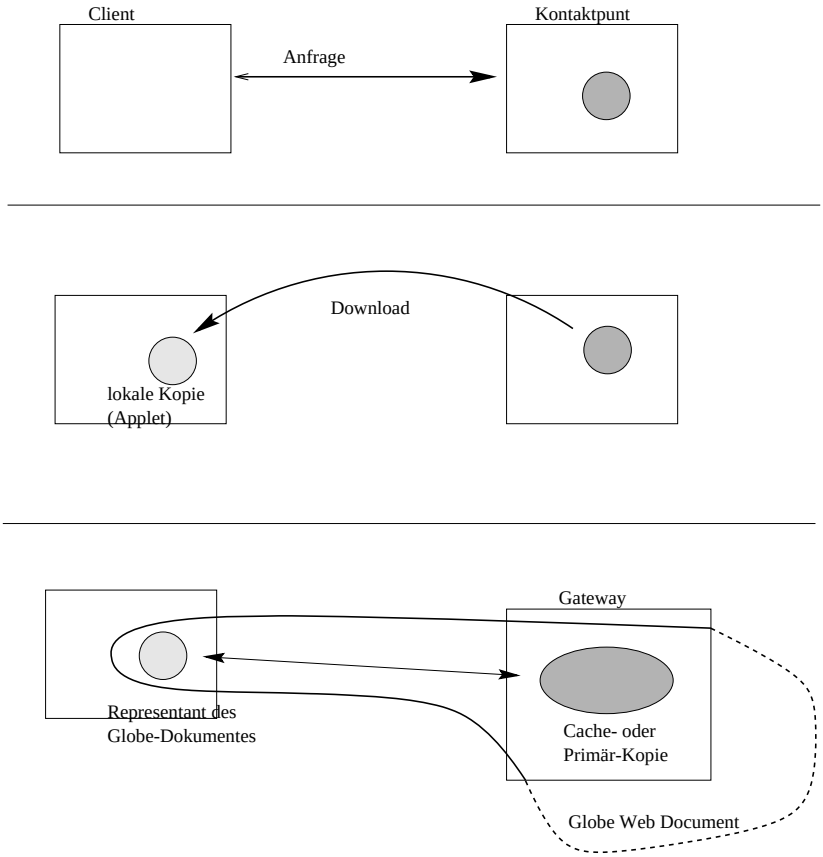


Abbildung 9: Anbindung an ein GlobDocs

6 Vergleichende Einordnung und Zusammenfassung

Nachdem einige Ansätze für verteilte, dynamische Objekte vorgestellt worden sind, soll nun deren Einordnung folgen. Wie in der Einleitung in Aussicht gestellt, werden sie nach den drei Kriterien räumliche und strukturelle Dynamik und nach deren Kommunikationsbedarf sortiert. So entsteht ein "Ordnungswürfel", der in Abbildung 10 skizziert ist.

Es bezeichnen:

1. Verteilte, dynamisch konstruierbare Objekte (Kap. 3)

2. statisch fragmentierte Objekte (Kap. 4.1)

3. Client-gesteuerte fragmentierte Objekte (Kap. 5.1)

4. Server-gesteuerte fragmentierte Objekte (Kap. 5.2)

Was bedeutet das nun im Einzelnen und welche Konzequenzen ergeben sich daraus für deren Einsatz ?

1. Das als erstes vorgestellte Konzept Gründers (Nr.1) setzt auf dem Dienstgeber/-nehmer-Modell auf. Der Server verfügt aber vor allem über hohe strukturelle Flexibilität, die sogar zur Wiederverwendung von Code in einer Klassenhierarchie benutzt werden kann. Nachteilig ist zum einen, daß die Lokalität der einzelnen Teile nach der Initialisierung unveränderbar bleibt: ein bestimmtes Chassis z.B. ist an seinen Ort gebunden, auch wenn es im Laufe seines Lebens den verschiedensten Objekten angehört. Zum anderen muß dem deutlich gesteigerten Kommunikationsbedarf Rechnung getragen werden. Es werden ja mindestens dreimal so viele Kommandos und Daten zwischen den Rechenknoten hin-

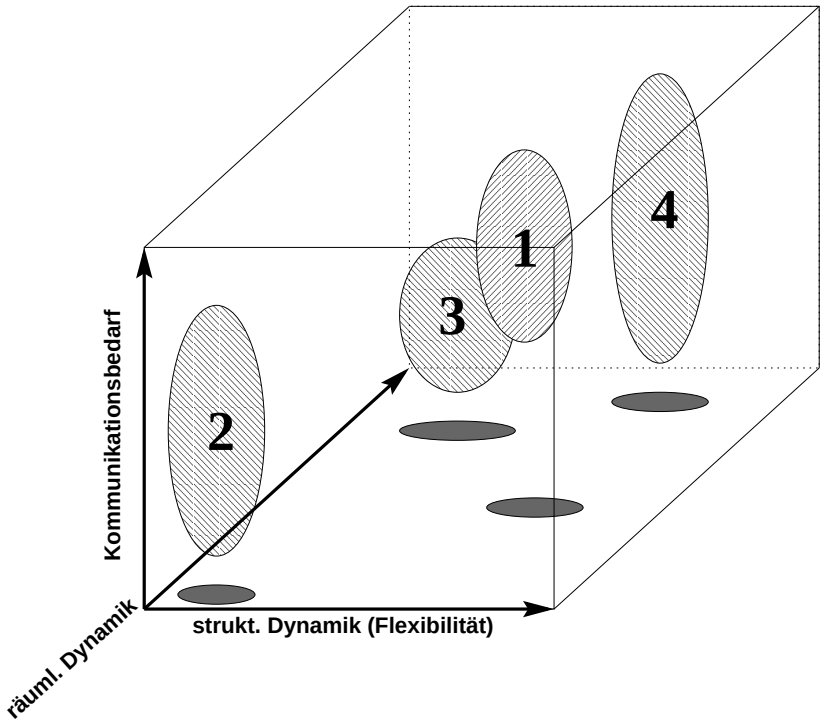


Abbildung 10: Einordnung der 4 Konzepte

und herbewegt als sonst.
Einsatzmöglichkeiten bieten sich also in schnellen LANs, in denen ein Client-Server-Modell (womöglich durch Hardware festgelegt) benutzt werden soll.

2. Statische Fragmente (Nr.2) geben die Dynamik zur Laufzeit zugunsten eines verringerten Datenverkehrs auf. Und noch weitere Vorteileergeben sich aus diesem Modell :

   - die einzelnen Fragmente können unabhängig voneinander weiterentwickelt oder neu erstellt werden,
   - die Architektur ist frei wählbar, die implementierungsspezifischen Einzelheiten werden vor dem Entwickler weitestgehend versteckt.

   Wie an Abschnitt 4.2 erläutert, kann sogar die Entscheidung über den richtigen Standort eines Fragments vom Computer getroffen werden, wodurch die Effektivität gesteigert werden kann. Dabei ergibt sich allerdings ein neues Problem für den Algorithmus: Die zu verteilende Soft- und Hardware muß ausreichend und richtig spezifiziert werden. Es gibt jedoch derzeit keine geeignete, standardisierte Beschreibungssprache.

3. Der Ansatz Client-gesteuerter Objekte in Abschnitt 5.1 basiert auf dem Client-Server-Ansatz beziehungsweise auf dem Code-on-Demand-Prinzip. Der neue Gedankengang, den Grosse begeht, nämlich Teile des Servers zum Client zu schieben, ist auf die Verwendung im Internet gerichtet. Den Kommunikationsverkehr bei gleichzeitiger verkleinerung der Verzögerung so niedrig wie möglich zu halten, war die zugrundeliegende Motivation. Dabei leidet konzequenterweise die strukturelle Dynamik, die Gründer et al. gewonnen hatten (nichts spräche dagegen, die beiden Techniken zu kombinieren). Grosses Ansatz ist dafür in punkto räumlicher Dynamik höher anzusiedeln, denn Nr.1 sieht kein Verschieben irgendwelcher Teile vor.
   Einsatzgebiete, sind alle Arten von verteilten Systemen, die fürs Internet optimiert werden müssen.

4. Tannenbaums GLOBE ähnelt einmal mehr den Fragmenten aus 4.1. Im Gegensatz zu diesen ist es aber ausdrücklich für eine Verwendung in stark frequentierten WWW-Seiten konstruiert worden, und integriert dabei sogar Mechanismen, die denen aus Abschnitt 5.1 entsprechen (diese sind hier aber nicht konfigurierbar). Da Globe Wert auf größmögliche Skalierbarkeit legt (wählbare Strategien für Verteilung, Caching und Replizierung), ist es in Hinblick auf beide Formen der Dynamik ganz nach vorne zu setzen. Beim Kommunikationsbedarf kann man desshalb keine genaue Aussage machen, weil es von der angewandten Struktur abhängt, wie viele Daten übertragen werden.

Abschließend kann man urteilen: verteilte, dynamische Objekte/ Komponenten bilden den Anfang einer neuen Software-Revolution. Noch fehlen verbindliche Standards, ohne die alles nur Insel- bzw. Nischenprodukte bleiben. Aber nur wenn sich alle an die Schnittstellen und Konventionen halten, können solche Systeme ihre volle Leistung erbringen.

In Zukunft werden vermutlich die meisten Systeme, die auf dem Markt bestehen wollen, Dynamik und Verteilung unterstützen müssen. Vieleicht basieren sie dann auf einem oder einer Kombination der gerade vorgestellten Konzepte.

Literatur

[Fran97] Michael Franz. Dynamic Linking of Software Components. *IEEE Computer*, März 1997, S. 74–81.

[GGKB98] H. Gründer, K. Geihs, T. Knape und F. Baumert. Dynamisch verteilte Objekte. *Informatik Forschung und Entwicklung* 1(13), 1998, S. 26–37.

[GrDR98] A. Grosse, S. Dolk und R. Ruggaber. Flexible Vermittlung von skalierbaren Dienstobjekten in verteilten Systemen. In *Java-Informationstage ´98*, 1998.

[GrGe96] H. Gründer und K. Geihs. Reuse and Inheritance in Distributed Object Systems. In *Of the Int. Workshop on Trends in Distributed Systems*, 1996.

[LaVe97] Robert Laddage und James Veitch. Dynamic Object Technology. *Communications of the ACM* 40(5), Mai 1997, S. 37–38.

[MGNS94] M. Makpangou, Y. Gourhant, J.-P. Le Narzhul und M. Shapiro. *Fragmented Objects for Distributed Abstractions*, S. 170–186. IEEE Computer Society Press. Editors: T. Casavant and M. Singhal Place: Los Alamos, 1994.

[PuJN98] Sandeep Purao, Hemant Jain und Derek Nazareth. Effective Distribution of Object-Oriented Applications. *Communications of the ACM* 41(8), August 1998, S. 100–108.

[STKS98] M. van Steen, A.S. Tannenbaum, I. Kuz und H.J. Sips. A Scalable Middelware Solution for Advanced Wide-Area Web Services. In *Middelware´98*, 1998.