Johann Wolfgang Goethe–Universität
Frankfurt am Main

DIPLOMARBEIT

am Fachbereich 20 (Informatik)

MICO – MICO is CORBA

Eine erweiterbare CORBA–Implementierung für Forschung und Ausbildung

vorgelegt von

Kay Römer

Betreuer: Prof. Dr. K. Geihs

Februar 1998

Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Literatur und Hilfsmittel angefertigt habe.

Frankfurt am Main, den 8. Februar 1998

Kay Römer

Danksagung

Für die Betreuung der Diplomarbeit möchte ich Herrn Prof. Dr. K. Geihs danken. Ihm und seinen Mitarbeitern gilt zusätzlicher Dank für die guten Arbeitsbedingungen an der Professur für verteilte Systeme und Betriebssysteme.

Herrn A. Puder danke ich für viele Diskussionen, seine Geduld mit mir und die ausgezeichnete Zusammenarbeit im Rahmen des Projektes. Er hat wesentliche Teile zu MICO beigesteuert.

Mein Dank gilt auch all denen, die MICO in einem frühen Stadium verwendet und aktiv zur Verbesserung und Weiterentwicklung beigetragen haben. Insbesondere sind dabei zu nennen: Kai–Uwe Sattler, Lars Doelle, Owen Taylor, Philippe Merle, Christian Becker, Christoph Gebauer, Torben Weis, Mathias Braun, sowie die Teilnehmer und Betreuer des CORBA–Praktikums im Wintersemester 1997/98.

Für die Bereitstellung von Rechnerressourcen möchte ich dem Mitarbeiter A. Heckwolf der Professur für Architektur und Betrieb verteilter Systeme und Telematik danken. Dadurch war es möglich, MICO auf verschiedenen Rechnertypen zu testen.

Besonderer Dank gilt auch all denjenigen, die frühe Versionen dieser Arbeit Korrektur gelesen und damit zu deren Verbesserung beigetragen haben, insbesondere sind dabei Josefine Schuhmann, Arno Puder, Christian Becker, Frank Römer und Michael Zapf zu nennen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ACE	Adaptive Communications Environment
BOA	Basic Object Adapter
CDR	Common Data Representation
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DCE	Distributed Computing Environment
DCE–CIOP	DCE Common Inter–ORB Protocol
DCOM	Distributed Componenent Object Model
DDCF	Distributed Document Component Facility
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
DSOM	Distributed System Object Model
DPE	Distributed Processing Environment
ESIOP	Environment–Specific Inter–ORB Protocol
FIFO	First In First Out
GIOP	General Inter–ORB Protocol
GNU	GNU is not Unix
GUID	Globally Unique Identifier
GNU GPL	GNU General Public License
IDL	Interface Definition Language
IIOP	Internet Inter–ORB Protocol
IOR	Interoperable Object Reference
IP	Internet Protocol
IR	Interface Repository

ISO	International Standards Organization
ITU	International Telecommunication Union
KI	Künstliche Intelligenz
LOA	Library Object Adapter
MAQS	Management for Adaptiv QoS–enabled Services
MICO	MICO is CORBA
NCCE	Native Computing and Communications Environment
OA	Object Adapter
ODP	Open Distributed Processing
ODL	Object Definition Language
OMA	Object Management Architecture
OMG	Object Management Group
OODB OA	Object–Oriented Database Object Adapter
ORB	Object Request Broker
OSF	Open Software Foundation
OSI	Open Systems Interconnection
POA	Portable Object Adapter
POS	Persistent Object Service
RIOP	Realtime Inter–ORB Protocol
RFP	Request for Proposal
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SII	Static Invocation Interface
SSI	Static Skeleton Interface
TCP	Transmission Control Protocol
TAO	The ACE ORB
TINA	Telecommunications Information Networking Architecture
UML	Uniform Modelling Language
URL	Uniform Resource Locator
WWW	World Wide Web

Kapitel 1

Einleitung

Motivation und Problemstellung

Organisatorische und technische Anforderungen vieler aktueller Problemstellungen an die Computertechnologie können häufig nicht mehr durch ein alleinstehendes *zentralisiertes System* erfüllt werden. Die dramatische Zunahme preiswert verfügbarer Netzwerkbandbreite, Rechen– und Speicherkapazität während des letzten Jahrzehnts hat es ermöglicht, diesen Problemstellungen durch Anwendung *verteilter Systeme* gerecht zu werden.

Gleichzeitig führt die inhärente Komplexität verteilter Systeme dazu, daß der Aufwand für Entwicklung und Wartung verteilter Anwendungen ungleich größer ist als der für Anwendungen auf zentralisierten Systemen. Daher muß der Entwickler bei der Lösung dieser Aufgaben durch sogenannte *Middleware* unterstützt werden. Diese verbirgt einen Teil der Komplexität verteilter Systeme und bietet Unterstützung bei den Aspekten, die nicht transparent gehandhabt werden können.

Mit der *Object Management Architecture (OMA)* hat die *Object Management Group (OMG)* einen offenen Standard für eine Middleware–Plattform geschaffen, die unabhängig von einem konkreten Produkt oder einer konkreten zugrundeliegenden Technologie ist. Kern der OMA ist ein *Object Request Broker (ORB)*, der durch die *Common Object Request Broker Architecture (CORBA)* definiert ist.

Die konkrete Anwendung der CORBA–Technologie erfordert eine Implementierung der CORBA–Spezifikation. Um möglichst großen Spielraum für verschiedene Anwendungsumgebungen zu lassen, definiert die Spezifikation aber nur die vom Anwender sichtbaren Schnittstellen und deren Semantik. Daher erfordert die Umsetzung der Spezifikation in eine Implementation einen erheblichen konzeptionellen und technischen Aufwand. Dieser Aufwand war bisher jedesmal aufs neue erforderlich, wenn CORBA in einem neuen Umfeld eingesetzt werden sollte.

Ziel dieser Arbeit ist daher die Enwicklung eines konzeptionellen Rahmenwerkes für eine *erweiterbare* CORBA–Implementierung, die mit (im Vergleich zu

einer vollständigen Neu–Implementierung der CORBA–Spezifikation) geringem Aufwand an neue Anwendungsumgebungen angepaßt werden kann. Im praktischen Teil der Arbeit werden diese Konzepte dann in Form von MICO (MICO Is CORBA) implementiert. Dabei handelt es sich um eine zur Version 2.0 der CORBA–Spezifikation konforme CORBA–Implementierung in C++, die vom Autor dieser Arbeit und Arno Puder vorgenommen wurde. MICO ist im Rahmen der *GNU General Public License* frei verfügbar.

Zielgruppe dieser Arbeit sind Projekte in Forschung und Ausbildung. Im Rahmen der Ausbildung soll der interne Aufbau von Middleware vermittelt werden können, indem gezielt einzelne Komponenten von Auszubildenden neu entwickelt und ausgetauscht werden. Forschungsvorhaben im CORBA–Umfeld können Nutzen aus der vorliegenden Arbeit ziehen, indem neue Komponenten zum System hinzugefügt werden, anstelle eine CORBA–Implementierung von Grund auf neu zu entwickeln.

Gliederung

Die Arbeit ist in sieben Kapitel untergliedert. Mit Kapitel 2 schließt sich eine Einordnung der Arbeit in das Umfeld verteilter Systeme an. Nach Vorstellung grundlegender Begriffe und Modelle wird dort ein Überblick über aktuelle Middleware–Architekturen und insbesondere CORBA gegeben. Abschließend wird beispielhaft die Architektur einer konventionellen CORBA–Implementierung vorgestellt. Kapitel 3 bis 6 sind der Hauptteil dieser Arbeit und stellen die Konzepte und das Design der erweiterbaren CORBA–Implementierung MICO vor. Jedes dieser Kapitel schließt mit einer Zusammenfassung, Bewertung und Diskussion von Alternativen ab. Kapitel 7 faßt die gesamte Arbeit zusammen und gibt einen Ausblick. Im Anhang finden sich eine Installationsanleitung und die Quelltexte von MICO.

Kapitel 2

Grundlagen

2.1 Verteilte Systeme

Die Fachliteratur enthält eine Reihe von Definitionen des Begriffs *verteiltes System*, die sich in einigen Details voneinander unterscheiden. Hier soll eine verallgemeinerte Definition als Grundlage zur Diskussion der verschiedenen Vorstellungen über ein verteiltes System dienen:

> *Ein verteiltes System ist ein informationsverarbeitendes System, das aus einer Menge eigenständiger Komponenten besteht, die mittels Kommunikation bei der Erfüllung eines angestrebten Zieles kooperieren.*

Diese Definition enthält eine Reihe von „Variablen", deren Werte die genaue Vorstellung von einem verteilten System beeinflussen:

- *System:* Man unterscheidet *offene* und *nicht offene* Systeme. Erstere sind dadurch gekennzeichnet, daß die Zusammenarbeit der Komponenten auf *Standards* basiert, die für jedermann zugänglich und allgemein akzeptiert sind. Jede am offenen verteilten System beteiligte Komponente muß sich gemäß dieser Standards verhalten. Darüber hinaus sind jedoch keine Vorab–Annahmen über die Komponenten eines offenen verteilten Systems möglich.

- *eigenständige Komponenten:* Die Eigenständigkeit der Komponenten eines verteilten Systems ist ein wichtiges Merkmal der Abgrenzung gegenüber *Mehrprozessor–Systemen*. Intuitiv bedeutet Eigenständigkeit, daß jede Komponente eines verteilten Systems selbst ein (nicht verteiltes) informationsverarbeitendes System darstellt. Tanenbaum [63] definiert die Komponenten als Prozessoren, wodurch auch Multiprozessorsysteme seine Definition eines verteilten Systems erfüllen. Streng genommen sind die Prozessoren eines solchen Systems aber nicht eigenständig, da sie losgelöst aus

dem Multiprozessorsystem kein funktionsfähiges informationsverarbeitendes System darstellen. Bapat [4] kommt der Forderung nach Eigenständigkeit näher, weil er Komponenten mit eigenem Speicher erwartet.

- *Kommunikation:* Die Art der Kommunikation ist eng mit der Eigenständigkeit der Komponenten verbunden. Während eng gekoppelte Systeme über gemeinsamen Speicher kommunizieren können, ist für autonome Komponenten ein Kommunikations–Netz nötig. Die Aussagen der Literatur über die Kommunikation zwischen den Komponenten bewegen sich zwischen nicht näher spezifiziert [63] und der konkreten Forderung nach einem Kommunikations–Netz [4]. In der Notwendigkeit zur Kommunikation unterscheiden sich verteilte Systeme von *zentralisierten Systemen.*

- *Kooperation:* Wesentliches Kennzeichen eines verteilten Systems ist die gemeinsame Bearbeitung einer Aufgabe, d.h. das verteilte System ist mehr als die bloße Summe seiner Komponenten. Darin unterscheiden sich verteilte Systeme von *vernetzten Systemen*, dessen Komponenten in einer eher losen Verbindung zueinander stehen und weitgehend unabhängig voneinander arbeiten.

Verteilte Systeme weisen unter anderem folgende mögliche Vorteile auf [63], [17]:

- Abbildung organisatorischer Strukturen auf die Struktur eines verteilten Systems,
- parallele Abarbeitung von Teilaufgaben in den Komponenten,
- gemeinsame Betriebsmittelnutzung durch die Komponenten,
- Ausfallsicherheit und Fehlertoleranz durch Replikation von Diensten,
- Wirschaftlichkeit durch bedarfsorientierte Dimensionierung des verteilten Systems,
- theoretisch unbegrenzte Rechenkapaziät durch Hinzunahme neuer Komponenten.

Mit der Verwendung verteilter Systeme sind unter anderem folgende Probleme verbunden:

- viele inheränt zentrale Aufgabenstellungen lassen sich schlecht auf die Struktur eines verteilten Systems abbilden,
- Rechenkapazität nimmt wegen Aufwand für Kommunikation nicht linear mit Zahl der Komponenten zu,

- Software für verteilte Systeme ist komplex, da sie mit der Verteiltheit und den daraus resultierenden Problemen umgehen muß,
- Bedarf nach gegenseitigem Schutz der an einem verteilten System beteiligten Interessensgruppen,
- zusätzliche Fehlerquellen durch die Verteilung,
- hoher Aufwand für Administration verteilter Systeme,
- Heterogenität durch Komponenten verschiedener Hersteller, verschiedene Hardware, Betriebssysteme, Programmiersprachen, etc.

Generelles Ziel bei der Anwendung verteilter Systeme ist die Ausnutzung der Vorteile und die Relativierung der Nachteile. Punkte von besonderem Interesse sind dabei:

- performante und effiziente Durchführung von verteilten Berechnungen,
- Verbergen der Komplexität verteilter Systeme vor dem Benutzer, bzw. Bereitstellung von geeigneten Mechanismen zum Umgang mit Problemen, die nicht transparent für den Benutzer gehandhabt werden können bzw. sollen,
- Fehlertoleranz durch Replikation von Diensten,
- Erweiterung verteilter Systeme um neue Komponenten zur Laufzeit.

2.2 Kooperations–Modelle

Kooperations–Modelle sind ein konzeptionelles Hilfsmittel zur Strukturierung von Berechnungen in mehr oder weniger unabhängige Komponenten, die bei der Lösung einer Aufgabenstellung zusammenarbeiten. Diese Komponenten können dann den Knoten des verteilten Systems zugeordnet werden. Die hier vorgestellten Modelle *Client/Server–Modell*, *Objektmodell* und *Agenten* sind Abstraktionen von Teilaspekten des realen Lebens.

Ein wichtiges Kriterium für die Bewertung der Modelle ist die *Mobilität* der Komponenten, d.h., ob und wie die Komponenten während ihrer Lebenszeit ihren Aufenthaltsort ändern können. Solch eine Änderung des Aufenthaltsorts bezeichnet man als *Migration*. Man unterscheidet zwischen *schwacher* und *starker* Mobilität. Schwach mobile Komponenten können frei *plaziert* werden, bevor sie ihre Arbeit aufnehmen. Im Gegensatz dazu können stark mobile Komponenten während ihrer gesamten Lebenszeit beliebig ihren Aufenthaltsort wechseln. Durch Mobilität wird es möglich, den Kommunikationsaufwand zwischen Komponenten zu reduzieren, indem die an der Kommunikation beteiligten Komponenten an den

gleichen Ort migrieren. Dabei muß abgewogen werden, ob der Aufwand für die Migration höher wiegt als der Aufwand für die Kommunikation.

Eng mit der Mobilität ist die Frage nach der *Lokalisierung* von Komponenten verbunden, d.h. wie der aktuelle Aufenthaltsort einer Komponente ausfindig gemacht werden kann. Man unterscheidet die *explizite* und *implizite* Lokalisierung. Explizite Lokalisierung zeichnet sich dadurch aus, daß vor dem Zugriff auf eine Komponente explizit bei einem Dienst der Aufenthaltsort nachgefragt werden muß. Bei der impliziten Lokalisierung gibt es für jede Komponente einen festen Aufenthaltsort. Migriert die Komponente, so verbleibt am ursprünglichen Aufenthaltsort ein *Stellvertreter*, der für die Weiterleitung von Anfragen an den neuen Aufenthaltsort der Komponente sorgt.

2.2.1 Client/Server–Modell

Das Client/Server–Modell [9] baut auf der aus dem realen Leben bekannten Beziehung zwischen Kunden und Dienstleistern auf: Dienstleister stellen eine Dienstleistung zur Verfügung, die von Kunden in Anspruch genommen werden kann.

Bei Verwendung des Client/Server–Modells wird eine Problemstellung durch zwei Arten von Komponenten modelliert (Klienten und Server), wobei die einmal bestimmte Rollenverteilung fest ist. Um einen Dienst zu verwenden, schickt ein Klient eine *Anforderung* an den Server. Der Server bearbeitet daraufhin die Anforderung und schickt eine *Antwort* zurück an den Klienten.

Damit Klienten verschiedene Server voneinander unterscheiden können, muß jeder Server einen eindeutigen *Identifizierer* besitzen. Man unterscheidet Server auch danach, ob sie über Zustandsinformationen verfügen, die zwischen Dienstnutzungen durch Klienten erhalten bleiben. Von *zustandslosen* Servern bereitgestellte Dienste hängen nur von den Parametern des aktuellen Dienstauftrages ab, während sie bei *zustandsbehafteten* Servern zusätzlich von den vorher durchgeführten Dienstaufträgen abhängen.

Das Modell macht keine Aussagen über Aufenthaltsorte von Klienten und Servern und gibt damit auch keine Anhaltspunkte für Mobilität.

2.2.2 Objektmodell für verteilte Systeme

Grundlage für das Objektmodell bildet die Abstraktion von realen (beispielsweise „Buch“) und abstrakten (beispielsweise „Vorlesung“) Gegenständen des alltäglichen Lebens. Objekte zeichnen sich durch *Passivität* aus, d.h., sie sind im Gegensatz zu Lebewesen nicht zu selbständigem Handeln fähig. Stattdessen werden von außen initiierte Operationen auf Objekten durchgeführt, die auch den Zustand des Objektes ändern können.

Bei Verwendung des Objektmodells wird eine Problemstellung durch eine Menge von Objekten modelliert, die gegenseitig Operationen auf sich ausführen. Bei der Ausführung einer Operation nimmt das aufrufende Objekt gemäß des

Client/Server–Modells die Rolle eines *Klienten* ein, während das aufgerufene Objekt die Rolle eines *Servers* übernimmt. Diese Rollenverteilung ist aber nur temporär für die Dauer der Ausführung einer Operation gültig.

Formal sind Objekte existierende Einheiten, die über eine *Identität*, einen *Zustand* und ein *Verhalten* verfügen [7]. Die Identität ist während der Lebenszeit eines Objektes unveränderlich und ermöglicht den Zugriff auf ein Objekt von außen.

Zustand und Verhalten eines Objektes werden nach außen hin durch eine *Schnittstelle* verborgen. Die Schnittstelle verfügt über Operationen, die durch an das Objekt abgeschickte *Nachrichten* aufgerufen werden können. Der Aufruf von Operationen ist die einzige Möglichkeit, auf Zustand und Verhalten eines Objektes von außen zuzugreifen. Durch diese *Kapselung* können die Internas eines Objektes unsichtbar für Klienten modifiziert werden. Die Semantik einer Operation wird durch eine *Signatur* repräsentiert, die neben einem eindeutigen Bezeichner Art und Menge von Ein– und Ausgabeparametern spezifiziert.

Gleichheit von Objekten bezüglich bestimmter Eigenschaften (beispielsweise Gleicheit bezüglich der Schnittstelle) definiert eine Äquivalenzrelation auf allen Objekten. Die durch diese Relation induzierten Äquivalenzklassen werden häufig auch als *Objekt–Klassen* oder einfach nur *Klassen* bezeichnet. Alle Objekte einer Klasse besitzen denselben *Typ*.

Die *Subtyp–Relation* setzt Klassen in Beziehung zueinander. Eine Klasse A ist Subtyp einer Klasse B, wenn überall dort, wo Objekte der Klasse B erwartet werden, „ohne Schaden“ auch Objekte vom Typ A verwendet werden können. Die Subtyp–Relation induziert eine hierarchische Struktur auf allen Klassen, wobei prinzipiell jede Klasse Nachfolger einer beliebigen Menge anderer Klassen und Vorgänger einer beliebigen Menge von Klassen sein kann. Zur Konstruktion von Klassen, die in einer Subtyp–Beziehung zueinander stehen, wird das Prinzip der *Vererbung* angewendet. Dabei werden bereits bestehende Klassen um neue Operationen erweitert. Für die Vorgänger bezüglich der Subtyp–Relation wird in Zusammenhang mit Vererbung der Begriff *Basis–Klasse* und für die Nachfolger der Begriff *abgeleitete Klasse* verwendet. Objekte einer Klasse mit mehreren Basisklassen nennt man *polymorph*, weil sie gleichzeitig mehr als einen Typ realisieren.

Für Objekte sind alle Arten von Mobilität denkbar, jedoch kann gemäß der passiven Eigenschaft von Objekten eine Migration nur von außen ausgelöst werden. Objekte entscheiden also nicht selbst, wann sie migrieren.

2.2.3 Agenten

Agenten liegt die Vorstellung von intelligenten Lebewesen zugrunde, die im Gegensatz zu Objekten zu selbständigem, aktiven Handeln in der Lage sind. Man unterscheidet *intelligente* und *mobile* Agenten.

Intelligente Agenten zeichnen sich nach [67] durch *Reaktionsfähigkeit*, *zielgerichtetes Handeln*, *Autonomie* und *Kooperationsfähigkeit* aus. Intelligente Agenten nehmen also ihre Umgebung wahr, passen sich an diese an und verfolgen durch selbständiges Handeln und Kooperation mit anderen Agenten ein vorgegebenes Ziel, ohne daß eine ständige Überwachung durch „höhere Intelligenz“ notwendig ist.

Unter mobilen Agenten versteht man Software–Komponenten, die eine oder mehrere bestimmte Aufgabe(n) lösen und währenddessen migrieren. Im Unterschied zu Objekten im Objektmodell sind Agenten in der Lage, selbständig die Entscheidung zur Migration zu treffen. Grundlage für diese Entscheidung ist eine Analyse von Umgebungsdaten wie beispielsweise Systemauslastung, Kommunikationsaufwand, etc.

2.3 Kommunikations–Modelle

Während die im vorhergehenden Abschnitt vorgestellten Kooperations–Modelle die Strukturierung von Berechnungen durch Zerlegung in zusammenarbeitende Komponenten erlauben, sind Kommunikations–Modelle konzeptionelle Hilfsmittel für die Kommunikation zwischen diesen Komponenten.

2.3.1 Nachrichtenübertragung

Nachrichtenübertragung ist ein einfacher Basismechanismus zur Kommunikation zwischen zwei oder mehr Komponenten, auf denen die in den nachfolgenden Abschnitten beschriebenen Verfahren aufsetzen.

Dabei unterscheidet man die Rolle des *Senders*, der eine Nachricht abschicken möchte, und die der *Empfänger*, die eine Nachricht empfangen möchten. Nachrichten werden in einer bestimmten Sprache verfaßt und unterliegen einer bestimmten Semantik.

Die Übertragung der Nachricht erfolgt entweder *ungepuffert* oder *gepuffert*. Im ersten Fall wird der Sender beim Abschicken einer Nachricht solange blockiert, bis der Empfänger bereit ist, die Nachricht entgegenzunehmen. Bei gepufferter Übertragung kann der Sender sofort nach dem Abschicken der Nachricht weiterarbeiten, weil die Nachricht solange zwischengespeichert wird, bis der Empfänger bereit ist, die Nachricht entgegenzunehmen.

Ein weiteres Mittel zur Klassifizierung von Methoden zur Nachrichtenübertragung ist die verwendete Art der *Adressierung*. Man unterscheidet *Unicast*, *Multicast* und *Broadcast*, bei denen die Nachricht an genau einen Empfänger, eine Menge von Empfängern bzw. an alle möglichen Empfänger übertragen wird.

Modelle zur Übertragung der Nachrichten über ein Kommunikations–Netz werden in der Regel in Schichten strukturiert, wobei jede Schicht unter Verwendung der darunterliegenden Schichten eine bestimmte Funktion erfüllt und den

darüberliegenden Schichten zur Verfügung stellt. Bekanntester Vertreter eines solchen Modells ist das von der ISO spezifizierte *Open Systems Interconnection Reference Modell (OSI–Modell)* [11].

2.3.2 Entfernter Prozeduraufruf

Der entfernte Prozeduraufruf (RPC) überträgt das Modell des lokalen Prozeduraufrufs auf den Fall, in dem sich Aufrufer und aufgerufene Prozedur auf verschiedenen Rechnerknoten eines verteilten Systems befinden. Der RPC dient unter anderem als Basis zur Kommunikation im Client/Server–Modell und im Objektmodell.

Da eine Prozedur über Ein– und Ausgabe–Parameter verfügt, müssen vor dem Prozeduraufruf die Eingabeparameter vom Knoten des Aufrufers zum Knoten der Prozedur übertragen werden. Nach Abarbeitung der Prozedur müssen die Ausgabeparameter auf umgekehrtem Wege zurück zum Aufrufer übertragen werden. Diese Übertragung der Parameter wird durch zwei Nachrichtenübertragungen in entgegengesetzter Richtung bewerkstelligt.

Kennzeichnend ist dabei die Verwendung sogenannter *Stellvertreter* (siehe Abschnitt 2.4 für Details) in der aufrufenden und aufgerufenen Komponente. Durch sie ist es möglich, bei Aufrufer und Aufgerufenem die Vorstellung zu erwecken, es handle sich um einen lokalen Prozeduraufruf. *Aufgerufenen–Stellvertreter* befinden sich im Aufrufer und verhalten sich aus Sicht des Aufrufers wie die aufgerufene, entfernte Prozedur. *Aufrufer–Stellvertreter* befinden sich im Aufgerufenen und verhalten sich gegenüber der aufgerufenen Prozedur wie der entfernte Aufrufer.

Obwohl der RPC damit einen Teil der Komplexität verteilter Systeme vor Aufrufern und Aufgerufenen verbirgt, gibt es doch einige signifikante Unterschiede zwischen lokalem und entfernten Prozeduraufruf:

Synchronisierung: Beim lokalen Prozeduraufruf ist der Kontrollfluß des Aufrufers mit dem Kontrollfluß der aufgerufenen Prozedur *synchronisiert*, da der Aufrufer solange blockiert wird, bis die Prozedur abgearbeitet ist. Diese Semantik kann auch vom RPC unterstützt werden, jedoch ist damit keine Parallelität zwischen Aufrufer und Aufgerufenem möglich. Der sogenannte *asynchrone* RPC ermöglicht es dem Aufrufer, während der Abarbeitung des RPC parallel weiterzuarbeiten. Der asynchrone RPC kann beispielsweise durch Verwendung sogenannter *Future*–Objekte [53] realisiert werden.

Trennung der Adreßräume: Während die Übergabe von Parametern *by–value* direkt vom lokalen auf den entfernten Fall übertragbar ist, bereitet die Übergabe von Zeigern oder Referenzen im entfernten Fall Probleme. Im synchronen Fall genügt es, die referenzierten Daten vor dem RPC vom Aufrufer zum Aufgerufenen und danach vom Aufgerufenen zum Aufrufer

zu übertragen (*copy/restore*), weil der Aufrufer während der Ausführung des RPC blockiert ist. Im asynchronen Fall muß garantiert werden, daß der Aufrufer während der Ausführung des RPC die lokalen Daten nicht modifiziert. Ein weiteres Problem birgt die Verwendung globaler Variablen.

Fehler: durch die Trennung der Addreßräume kann es zu Fehlern kommen, die im Fall eines lokalen Prozeduraufrufes nicht möglich sind (Fehler bei der Nachrichtenübertragung, Ausfall der prozedurausführenden Komponente, etc). Für RPCs sind daher folgende Aufrufsemantiken geläufig:

- *best-effort:* keine oder beliebig häufige Ausführung des RPC; keine besonderen Maßnahmen notwendig
- *at–least–once:* RPC wird mindestens einmal ausgeführt oder der Benutzer erhält eine Fehlermeldung; der Aufrufer muß den RPC dazu zwischenspeichern und gegebenenfalls wiederholen
- *at–most–once:* RPC wird höchstens einmal ausgeführt oder der Benutzer erhält eine Fehlermeldung; der Aufgerufene muß Duplikaterkennung einsetzen
- *exactly–once:* RPC wird genau einmal ausgeführt oder der Benutzer erhält eine Fehlermeldung; Kombination der Maßnahmen für *at–most–once* und *at–least–once*

Das Bewußtsein, daß sich die Komplexität verteilter Systeme nicht vollständig vor dem Benutzer verbergen läßt, hat zur Definition sogenannter *Dienstqualitäten* geführt. Darunter versteht man nach [5] all die Eigenschaften einer Beziehung zwischen Dienstnutzer und Diensterbringer, die nicht Bestandteil des erbrachten Dienstes selbst sind. Beispielsweise sind die oben genannten Aufrufsemantiken (*best–effort*, *at–least–once*, etc.) Dienstqualitäten.

2.3.3 Kontinuierliche Datenströme

Während die beim RPC zwischen den Komponenten übertragenen Nachrichten *diskret* sind, bieten Datenströme die Möglichkeit zur Übertragung *kontinuierlicher*, potentiell unendlicher Ströme von Daten.

Von besonderer Bedeutung ist der Einsatz kontinuierlicher Datenströme unter Echtzeitbedingungen. Dabei werden im Gegensatz zur herkömmlichen Nachrichtenübertragung nicht nur Anforderungen an die Übertragung der Nachricht als Ganzes gestellt. Vielmehr müssen während der gesamten Übertragung bestimmte Dienstqualitäten eingehalten werden:

- *Rechtzeitigkeit:* die Daten treffen rechtzeitig beim Empfänger ein
- *Synchronität:* es tritt keine zeitliche Verzögerung zwischen parallel übertragenen Datenströmen ein

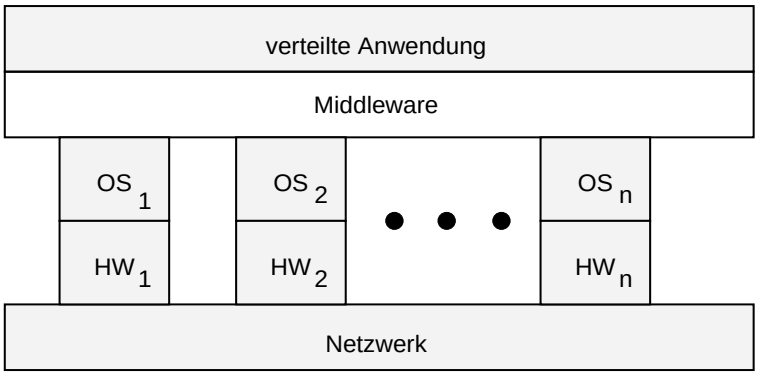


Abbildung 2.1: Middleware

Eine Standardanwendung für kontinuierliche Datenströme ist die Übertragung von Audio– und Videodaten. Wird dabei die Rechtzeitigkeit nicht garantiert, so kommt es zu Stockungen in Bild oder Ton. Bei nicht gewährleisteter Synchronität kommt es zu zeitlichen Verzögerungen zwischen Bild und Ton.

2.4 Middleware

Die Bewältigung der Komplexität verteilter Systeme erfordert einen erheblichen Aufwand bei der Entwicklung von Anwendungen für verteilte Systeme. Da jedoch viele der mit der Komplexität verteilter Systeme verbundenen Probleme bei vielen Anwendungen gleichermaßen auftreten, ist es sinnvoll, dem Benutzer die Lösung dieser Probleme durch Bereitstellung geeigneter Hilfsmittel abzunehmen oder zu erleichtern. Solche Hilfsmittel in Form von Software nennt man *Middleware* (deutsch: Verteilungsplattform), weil sie sich, wie in Abbildung 2.1 gezeigt, logisch zwischen verteilter Anwendung und verteiltem System befindet.

Generell unterstützt Middleware den Anwender dabei, die in Abschnitt 2.1 genannten Ziele bei der Anwendung verteilter Systeme zu erreichen. Als wichtige Details sollte Middleware dazu möglichst viele der folgenden Eigenschaften aufweisen:

- *Verteilungs–Transparenz:* kein Unterschied zwischen Zugriff auf lokale und entfernte Komponenten
- *Fehler–Transparenz:* automatisch korrigierbare Fehler werden für den Anwender unsichtbar behoben
- *Migrations–Transparenz:* Komponenten können unsichtbar für Nachfrager ihren Aufenthaltsort ändern
- *Programmiersprachen–Transparenz:* transparente Zusammenarbeit von

Komponenten, die in verschiedenen Programmiersprachen implementiert sind

- *Betriebssystem–Transparenz:* transparente Zusammenarbeit von Komponenten, die auf Rechnern mit verschiedenen Betriebssystem ausgeführt werden
- *Hardware–Transparenz:* transparente Zusammenarbeit von Komponenten, die auf Rechnern mit verschiedener Hardware ausgeführt werden
- *Hersteller–Transparenz:* transparente Zusammenarbeit von Middleware–Systemen verschiedener Hersteller

Zur Bereitstellung dieser Eigenschaften unterstützt Middleware den Anwender bei der *Entwicklung* und während der *Laufzeit* von Anwendungen für verteilte Systeme:

- Entwicklung (u.a.):
  - Schnittstellenbeschreibungssprachen
  - Kode–Generatoren
- Laufzeit (u.a.):
  - Management von Komponenten der verteilten Anwendung
  - Kommunikation zwischen Komponenten der verteilten Anwendung

Ein Middleware–System basiert in der Regel auf einem der in Abschnitt 2.2 vorgestellten Kooperationsmodelle und bietet damit einen Mechanismus zur Strukturierung verteilter Berechnungen durch Verwendung bestimmter Komponenten. Bei auf dem Objektmodell basierenden Middleware–Plattformen wie CORBA [41] oder DCOM [50] sind diese Komponenten Objekte.

Zur *Entwicklung* dieser Komponenten bieten Middleware–Systeme geeignete Hilfsmittel. Beispielsweise werden die von Komponenten nach außen angebotenen Schnittstellen oftmals unabhängig von bestimmten Programmiersprachen mittels sogenannter *Schnittstellenbeschreibungssprachen* spezifiziert. *Kode–Generatoren* erzeugen daraus Schablonen in einer bestimmten Programmiersprache, die dann vom Programmierer vervollständigt werden müssen.

Middleware–Systeme stellen außerdem ein *Laufzeit–System* zur Verfügung, das unter anderem für das *Management* des Lebenszykluses der mit Hilfe der Entwicklungsumgebung erzeugten Komponenten sorgt. Darunter fallen Aufgaben wie:

- Erzeugung/Zerstörung von Komponenten

- Aktivierung/Deaktivierung von Komponenten
- Verschieben von Komponenten (Migration)
- Kopieren von Komponenten

Ein besonders wichtiges Management–Detail ist das Herstellen von *Bindungen* zwischen Komponenten, um eine nachfolgende Kommunikation zwischen diesen Komponenten zu ermöglichen. Bindungen können entweder *statisch* während der Entwicklung der Komponenten oder *dynamisch* zur Laufzeit durch Verwendung von *Tradern* oder *Namensdiensten* hergestellt werden.

Nach Zustandekommen einer Bindung können die Komponenten miteinander *kommunizieren*. Dazu unterstützt Middleware ein oder mehrere der in Abschnitt 2.3 vorgestellten Kommunikations–Modelle.

2.5 Überblick über Verteilungs–Plattformen

In den letzten Jahren sind eine Reihe von Verteilungs–Plattformen wie DACNOS [15], DCE [44], CORBA [41], DCOM [50], TINA [12] oder Java RMI [60] spezifiziert und implementiert worden. Daneben gehen von der ISO unter dem Namen *Open Distributed Processing (ODP)* [23, 24, 25, 26] Standardisierungsbestrebungen aus, bei denen es sich weniger um eine bestimmte Middleware–Architektur, sondern um ein auf dem Objektmodell basierendes, konzeptionelles Rahmenwerk für Berechnungen in verteilten Systemen handelt.

In diesem Abschnitt werden zwei aktuelle Middleware–Architekturen vorgestellt, die auf auf dem Objektmodell basieren. Der Schwerpunkt liegt dabei auf CORBA, da es die Grundlage dieser Arbeit bildet.

2.5.1 Object Management Architecture (OMA)

Einer der wichtigsten Verfechter objektorientierter Modellierung im Umfeld verteilter Systeme ist die *Object Management Group (OMG)*. Ihr erklärtes Ziel ist die Schaffung einer objektorientierten Infrastruktur als Grundlage für die Informationsverarbeitung der Zukunft.

Bei der OMG handelt es sich um ein 1989 gegründetes internationales Konsortium, dem inzwischen weltweit 700 Mitglieder aus dem Bereich der Informations–Technologie angehören. Der Entwurf von Spezifikationen orientiert sich dabei an bereits existierender Technologie der Mitglieder. Dazu werden von der OMG Anfragen an die Mitglieder in Form sogenannter *Request for Proposals (RFP)* gestellt. Von den Mitgliedern daraufhin eingereichte Vorschläge werden diskutiert und schließlich durch Abstimmung angenommen oder verworfen.

Auf Basis dieses Vorgehens wurde von der OMG die *Object Management Architecture (OMA)* [39] spezifiziert, eine Plattform zur Entwicklung verteilter, objektorientierter Anwendungen. Im Gegensatz zu vielen anderen Architekturen

handelt es sich dabei nicht um eine Implementierung oder gar ein Produkt, sondern um eine frei verfügbare Spezifikation, die keine Vorschriften über die für eine Implementierung zu verwendende Technologie macht.

Kernpunkte der OMA sind ein abstraktes *Objektmodell* und die sogenannte *Referenz–Architektur*. Das *Objektmodell* der OMA spezifiziert die Konzepte, die Objekten im Rahmen der OMA zugrundeliegen. Es ähnelt stark dem in Abschnitt 2.2.2 beschriebenen Modell, in dem Objekte Entitäten mit Identität, Zustand und Verhalten sind, wobei Zustand und Verhalten nach außen durch eine Menge von Operationen in Form einer Schnittstelle repäsentiert werden.

Das OMA–Objektmodell unterscheidet zwischen *Objekt–Semantik* und *Objekt–Implementierung*. Die Objekt–Semantik beschreibt die nach außen hin für Klienten sichtbaren Eigenschaften von Objekten, während sich die Objekt–Implementierung mit den Konzepten zur Realisierung von Objekten befaßt. Der Schwerpunkt der Spezifikation wird dabei auf die Objekt–Semantik gelegt. Die Objekt–Implementierung wird nur grob umrissen, um maximale Freiheit für die Implementierung von Objekten zu lassen.

Das Objektmodell der OMA [39] ist *abstrakt* in dem Sinne, daß es nicht direkt zur Konstruktion von Anwendungen verwendet werden kann. Um das Objektmodell zu *konkretisieren* und damit anwendbar zu machen, sind folgende Mechanismen vorgesehen:

1. *Detailierung* durch genauere Vorgaben; beispielsweise durch Vorgabe einer Sprache zur Definition von Typen

2. *Bevölkerung* durch Einführung spezieller Instanzen; beispielsweise spezieller Objekte, Operationen oder Typen

3. *Einschränkung* durch Entfernen von Teilen des Modells oder durch Einschränkung der Verwendbarkeit von Teilen des Modells

Beispielsweise geht durch Anwendung dieser Mechanismen das unten vorgestellte konkrete CORBA–Objektmodell aus dem abstrakten OMA–Objektmodell hervor.

Während das Objektmodell Objekte im Sinne der OMA charakterisiert, definiert das *Referenzmodell* Beziehungen zwischen Objekten. Abbildung 2.2 zeigt die Struktur des Referenzmodells. Zentrale Komponente ist der *Object Request Broker (ORB)* [41], der als Software–Bus die Kommunikation zwischen Objekten vier verschiedener Kategorien ermöglicht:

Object Services: Horizontale[1] System–Dienste, die von vielen Anwendungen benötigt werden (beispielsweise Namensdienst) [35].

[1]Während *horizontale* Dienste bereichsübergreifend eingesetzt werden können, werden *vertikale* Dienste nur für ganz bestimmte Einsatzbereiche benötigt.

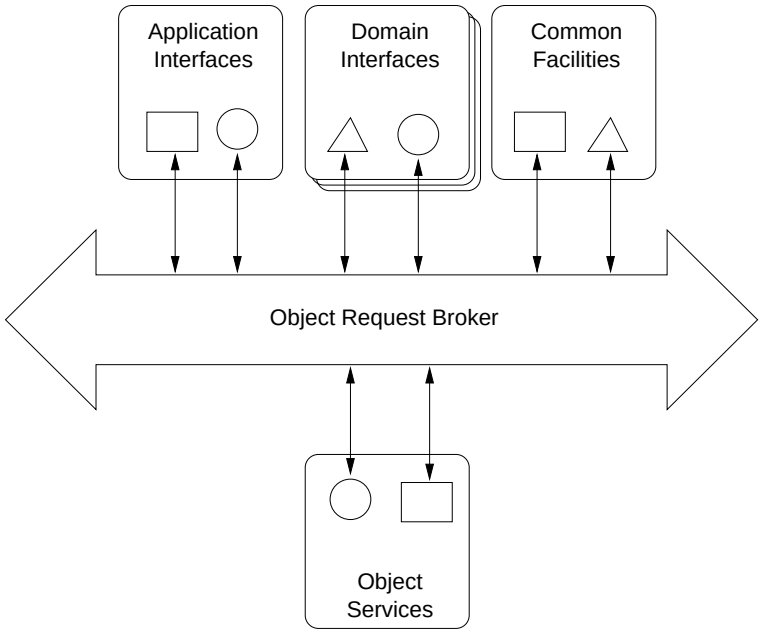


Abbildung 2.2: OMA–Referenz–Architektur

Common Facilities: Horizontale Endanwender–Dienste, die von vielen Anwendungen benötigt werden (beispielsweise stellt die *Distributed Document Component Facility (DDCF)* [34] auf der Basis von OpenDoc [10] einen Mechanismus zur Definition, Handhabung und Präsentation verteilter, zusammengesetzter Dokumente bereit).

Domain Interfaces: Vertikale Dienste für spezielle Anwendungsbereiche (beispielsweise für Medizin, Telekommunikation, etc).

Application Interfaces: Anwendungs–Objekte, die im Gegensatz zu den anderen Kategorien nicht durch die OMG spezifiziert werden.

Durch *Implementierung* dieser von der OMG oder vom Anwender spezifizierten *Schnittstellen* lassen sich sogenannte *Objekt–Frameworks* erstellen, die für einen bestimmten Anwendungsbereich eine Art verteilte Klassenbibliothek darstellen. Obwohl sich aus Sicht des Anwenders jedes der Objekte eines Frameworks einer der vier obigen Kategorien zuordnen läßt, können die Objekte durch Anwendung geeigneter Mechanismen (beispielsweise Vererbung) zusätzlich Schnittstellen aus Kategorien implementieren, die auf einer niedrigeren Abstraktionsstufe stehen.
Abbildung 2.3 zeigt die Struktur eines solchen Frameworks. Objekte bestehen darin aus einem Kern, der die Implementierung darstellt, und einer Schale, die die Schnittstellen des Objektes repräsentiert.

Der als Software–Bus zwischen den Objekten fungierende ORB ist durch die *Common Object Request Broker Architecture (CORBA)* spezifiziert. Grundlage

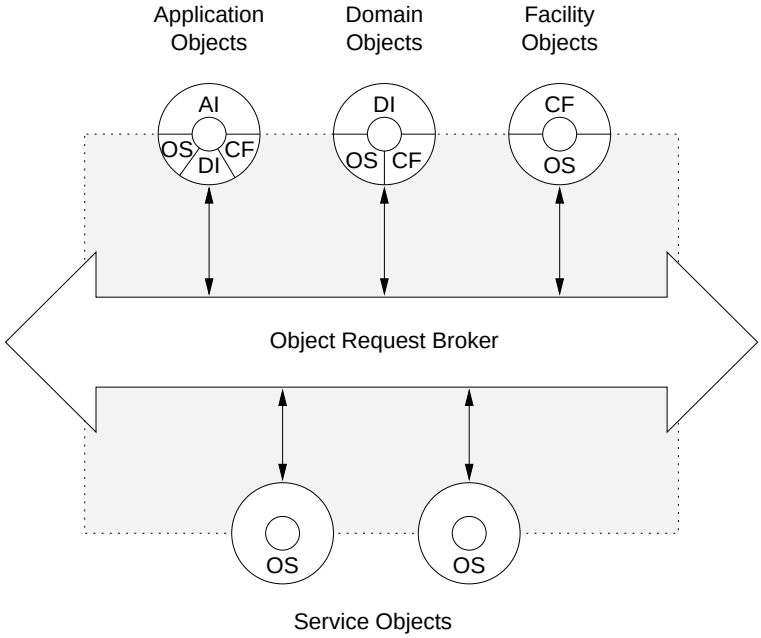


Abbildung 2.3: Objekt–Framework

von CORBA ist das CORBA–Objektmodell, das eine konkrete Instanz des abstrakten OMA–Objektmodells ist. Durch Anwendung der Regeln zur Konkretisierung des OMA–Objektmodells definiert das CORBA–Objektmodell die Menge der zulässigen Typen. Wie in Abbildung 2.4 gezeigt werden neben Objektreferenzen die in vielen Programmiersprachen üblichen Basistypen und zusammengesetzten Typen zugelassen. Erwähnenswert sind der Typ `Any`, der Werte beliebigen Typs (einschließlich sich selbst) aufnehmen kann, und der Typ `Except`, der eine *Ausnahme* repräsentiert, die während der Ausführung einer Operation auftreten kann.

Außerdem definiert das CORBA–Objektmodell Signatur und Aufrufsemantik von Operationen einer Schnittstelle. Danach besteht die Signatur einer Operation aus:

- *Aufrufsemantik:* legt fest, welche Aufrufsemantik für die Operation verwendet wird (möglich sind *best–effort* oder *at–most–once*, siehe Abschnitt 2.3.2)
- *Ergebnistyp*
- *Operationsname*
- *Parameter–Liste:* jeder Parameter verfügt über einen Typ und ein *Tag*, das

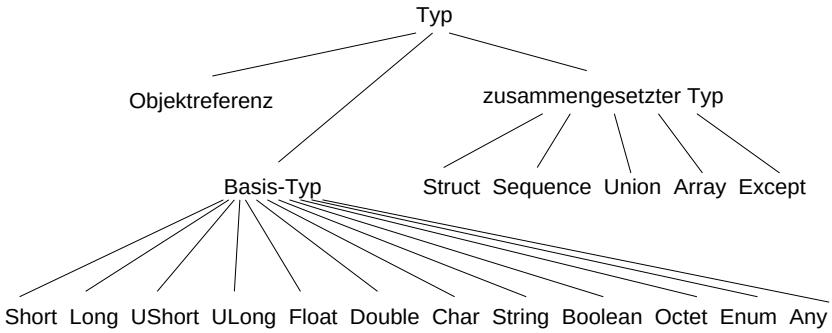


Abbildung 2.4: Typen im CORBA–Objektmodell

den Parameter als Eingabe– oder Ausgabe–Parameter oder beides kennzeichnet

- *Exception–Liste:* optionale Liste von Ausnahme–Typen, die bei der Bearbeitung der Operation auftreten können
- *Kontext–Liste:* optionale Liste von Kontext–Informationen

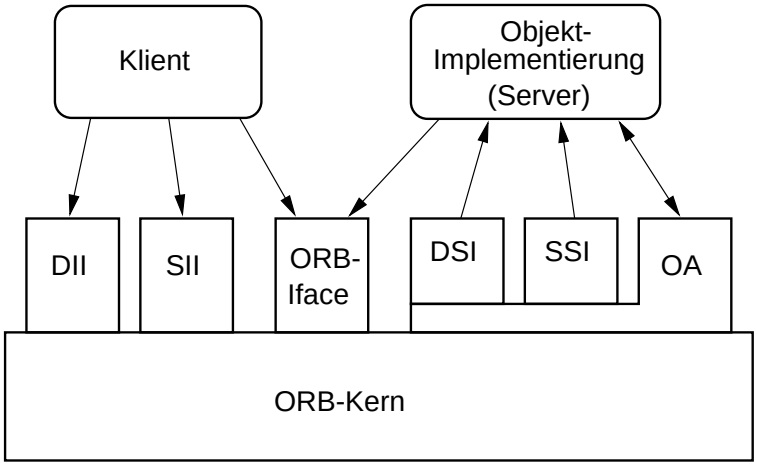
Das eben kurz skizzierte CORBA–Objektmodell spiegelt sich direkt in der *Interface Definition Language (IDL)* wider, mit der Typen des Objektmodells und insbesondere die Schnittstellen von Objekten formal beschrieben werden können. Neben der IDL wird in den nun folgenden Abschnitten näher auf die einzelnen Komponenten der CORBA–Architektur eingegangen. Abbildung 2.5 gibt einen Überblick über die wichtigsten Komponenten.

IDL

Die *Interface Defition Language (IDL)* dient zur Spezifikation der Schnittstellen von Objekten unabhängig von einer bestimmten Programmiersprache. Damit ist die IDL Grundlage für die Trennung von Schnittstelle und Implementierung eines Objektes.

CORBA–IDL ist eine *deklarative* Sprache, d.h. sie enthält keine algorithmischen Konstrukte zur Bildung von Schleifen, Verzweigungen, etc. Ihre Syntax lehnt sich stark an die von C++ an, verfügt jedoch über einige zusätzliche Konstrukte, um der Anwendung in einer verteilten Umgebung gerecht zu werden (beispielsweise zur Kennzeichnung von Parametern als Eingabe– oder Ausgabe–Parameter oder beides).

Die Konstrukte der CORBA–IDL ermöglichen es, zum CORBA–Objektmodell konforme Typen zu definieren. Um bereits vorhandene Typen bei der Konstruktion neuer wiederzuverwenden, stellt IDL den Mechanismus der *Schnittstellen–Vererbung* zur Verfügung. Im Gegensatz zur in vielen objektorientierten Programmiersprachen üblichen *Implementations–Vererbung*,



DII	Dynamic Invocation Interface
SII	Static Invocation Interface
DSI	Dynamic Skeleton Interface
SSI	Static Skeleton Interface
OA	Objekt-Adapter

Abbildung 2.5: CORBA

bei der Schnittstellen und deren Implementierungen vererbt werden, erlaubt Schnittstellen–Vererbung nur die Wiederverwendung von Schnittstellen. Um bei der Konstruktion neuer CORBA–Objekte dennoch bereits existierenden Kode wiederverwenden zu können, muß auf geeignete Mechanismen der Implementationssprache (beispielsweise Vererbung, Delegation, Aggregation) zurückgegriffen werden.

Zur Zusammenfassung von verwandten Schnittstellen erlaubt IDL die Definition sogenannter *Module*. Da die geschachtelte Anwendung von Modulen möglich ist, verfügt CORBA–IDL über einen hierarchischen Namensraum.

IDL Sprachanbindungen

Die Abbildung von IDL Schnittstellen auf eine bestimmte Programmiersprache wie C++ definieren die sogenannten *IDL–Sprachanbindungen*. Derzeit hat die OMG Sprachanbindungen für C, C++, Smalltalk, Ada 95, COBOL und Java definiert. Sogenannte *IDL–Compiler* automatisieren die Verwendung dieser Sprachanbindungen, indem sie eine IDL–Spezifikation einlesen und entsprechenden Kode in einer der oben genannten Programmiersprachen erzeugen.

Die einfachen und zusammengesetzten Datentypen der CORBA–IDL (siehe auch Abbildung 2.4) werden auf entsprechende Datentypen der Zielsprache abgebildet. Jede IDL Schnittstelle wird auf zwei *Stellvertreter* abgebildet: einen *Stub* und ein *Skelett*. Ähnlich wie beim RPC befindet sich der Stub im Klienten und

verhält sich gegenüber einem Aufrufer wie das entfernte CORBA–Objekt. Das Skelett befindet sich im Server und verhält sich gegenüber dem lokalen CORBA–Objekt wie der entfernte Aufrufer.

Aufbau von Stubs und Skeletten ist stark von der verwendeten Programmiersprache abhängig. In objektorientierten Programmiersprachen werden Objekte verwendet, während in prozeduralen Programmiersprachen Mengen von Funktionen bzw. Prozeduren die Rolle von Stub und Skelett übernehmen.

Hauptaufgabe eines Stubs ist die Übergabe von Methodenaufrufen an den ORB–Kern durch Verwendung eines Aufruf–Adapters. Skelette nehmen Methodenaufrufe von einem Objekt–Adapter entgegen und leiten sie an die Objekt–Implementierung weiter.

Interface Repository

Das *Interface Repository (IR)* ermöglicht den Zugriff auf Typinformationen zur Laufzeit. Die im IR abgelegten Daten entsprechen dabei den in einer IDL Spezifikation enthaltenen Informationen. Durch Verwendung des IR wird es Klienten möglich, Operationen auf Objekten aufzurufen, deren Schnittstellen bei der Entwicklung des Klienten noch nicht bekannt waren.

Analog zum hierarchischen Namensraum der CORBA–IDL sind die Informationen im IR in Form einer Hierarchie abgelegt. Der Zugriff auf einen bestimmten Eintrag dieser Hierarchie kann entweder durch *Browsen* [48] oder durch Verwendung sogenannter *Repository ID*s geschehen. Bei letzteren handelt es sich um Zeichenketten mit einem speziellen Format, die eindeutig einen Knoten in der Hierarchie des IR identifizieren. Wird beispielsweise die IDL–Spezifikation

```
// IDL
module foo {
    interface bar {
        ...
    };
};
```

im Interface Repository abgelegt, so kann mittels der Repository ID

```
IDL:foo/bar:1.0
```

auf den Knoten des IR zugegriffen werden, der die Schnittstelle `bar` in der Version 1.0 repräsentiert.

ORB–Kern

Der ORB–Kern überträgt Methodenaufrufe von Klienten zu Servern, die sich im gleichen Adreßraum, in verschiedenen Adreßräumen auf einem Rechner oder auf

verschiedenen Rechnern befinden können. Er sorgt somit dafür, daß die Kommunikation zwischen Objekten in einer verteilten Umgebung transparent für die Objekte möglich ist.

Neben einer direkten Schnittstelle zum Anwender verfügt der ORB–Kern über nicht durch den CORBA–Standard spezifizierte Schnittstellen zu Aufruf–Adaptern und Objekt–Adaptern.

Kapitel 3 gibt einen detaillierten Überblick über die Funktionalität und das Design des ORB–Kerns in MICO.

Aufruf–Adapter

Klienten benutzen Aufruf–Adapter indirekt über Stub–Objekte oder auch direkt, um Methodenaufrufe an den ORB–Kern zu übergeben. Aufruf–Adapter sind vom ORB–Kern getrennte Komponenten, weil die zur Auslösung eines Methodenaufrufs notwendige Funktionalität stark variieren kann. Für bestimmte Anforderungen kann so ein spezieller Aufruf–Adapter verwendet werden.

CORBA sieht zwei verschiedene Typen von Aufruf–Adaptern vor: das *Static Invocation Interface (SII)* wird niemals direkt vom Anwender, sondern nur indirekt durch vom IDL Compiler generierte Stubs verwendet. Da das SII keine für den Anwender sichtbare Schnittstelle hat, macht die CORBA–Spezifikation keine weiteren Vorgaben für dessen Schnittstelle zu den Stubs und zum ORB–Kern. Im Gegensatz dazu ist das *Dynamic Invocation Interface (DII)* für die direkte Verwendung durch den Anwender vorgesehen. Unter Verwendung des Interface Repository und des DII können Klienten Methoden auf Objekten aufrufen, deren Schnittstellen während der Entwicklung des Klienten nicht bekannt waren.

In Kapitel 6 wird näher auf Eigenschaften und Funktionalität von Aufruf–Adaptern eingegangen. Außerdem wird das Design eines Aufruf–Adapters für MICO vorgestellt.

Objekt–Adapter

In Analogie zu Aufruf–Adaptern stellen Objekt–Adapter die Verbindung zwischen ORB–Kern und Objekt–Implementierungen her. Auch sie sind vom ORB–Kern getrennte Komponenten, weil verschiedene Arten von Objekt–Implementierungen verschiedene Anforderungen an Objekt–Adapter stellen.

Objekt–Adapter verwalten den Lebenszyklus von CORBA–Objekten und sorgen für die Ausführung von Methodenaufrufen. Dazu werden die Objekt–Implementierungen über sogenannte *Skeleton Interfaces* an den Objekt–Adapter gebunden. Geläufig sind *Static Skeleton Interfaces (SSI)* und *Dynamic Skeleton Interfaces (DSI)*. Während das SSI in Analogie zum SII zur Anbindung von durch den IDL Compiler generierten Skeletten an den Objekt–Adapter dient, ermöglicht das DSI *generische* Server, die Methodenaufrufe auf Objekten handhaben können, deren Schnittstelle bei der Entwicklung des Servers nicht bekannt

war. Eine wichtige Anwendung des DSI sind *Bridges* (siehe Abschnitt 5.2.2).

Kapitel 5 geht im Detail auf Funktionalität und Eigenschaften verschiedener Objekt–Adapter, sowie das Design des *Basic Object Adapter (BOA)* in MICO ein.

Interoperabilität

Um die Zusammenarbeit verschiedener CORBA–Produkte zu garantieren, spezifiziert der CORBA–Standard sogenannte *Inter–ORB–Protokolle*, die von jeder konformen CORBA–Implementierung bereitgestellt werden müssen. Beispielsweise muß jede CORBA 2 konforme Implementierung das *Internet Inter–ORB Protocol (IIOP)* unterstützen.

Auf das der Interoperabilität in CORBA zugrundeliegende Modell, die Inter–ORB–Protokolle und deren Realisierung in MICO wird in Kapitel 4 eingegangen.

2.5.2 Distributed Component Object Model (DCOM)

Das *Distributed Component Object Model (DCOM)* [50] von Microsoft ist ebenfalls eine auf einem Objektmodell basierende Middleware–Plattform. Während es sich bei der OMA der OMG um eine Spezifikation handelt, stellt DCOM ein konkretes Produkt dar. Erst in jüngster Zeit wurde die Architektur von DCOM offengelegt und an die Active Group [2] übergeben. Diese Offenlegung ermöglichte DCOM den Einzug in heterogene Umgebungen heraus aus dem homogenen Windows–Umfeld. Beispielsweise wird DCOM im Moment von der Software AG [58] auf verschiedene Unix–Plattformen portiert.

Obwohl DCOM wie die OMA auf der Objekt–Technologie basiert, gibt es doch erhebliche Unterschiede zwischen den DCOM und OMA bzw. CORBA zugrundeliegenden Objektmodellen. Beispielsweise kann ein Objekt in DCOM über mehrere voneinander unabhängige Schnittstellen verfügen. Zur Wiederverwendung bereits existierender Schnittstellen kann Delegation oder Aggregation zur Anwendung kommen. Bei der Aggreration mehrerer bereits existierender Schnittstellen entsteht ein neues Objekt mit mehreren Schnittstellen. Diese Form der Wiederverwendung ist flexibel, weil Beziehungen zwischen Objekten dynamisch zur Laufzeit hergestellt und modifiziert werden können, während Vererbung auf statischen Beziehungen zwischen Schnittstellen basiert. Oft ist die Modellierung mittels Vererbung aber einfacher, weil sie in natürlicher Weise *ist–ein* Beziehungen und damit Polymorphie zwischen Objekten widerspiegelt.

Schnittstellen werden in DCOM durch weltweit eindeutige, automatisch generierte *Globally Unique Identifiers (GUID)* referenziert. Im Vergleich zu von Menschen vergebenen Namen sollen sie verhindern, daß Schnittstellen verschiedenen Typs den selben Bezeichner tragen (*Name–Clash*). Gleichzeitig fehlt dadurch aber jeder inhaltliche Bezug zwischen Schnittstelle und GUID. CORBA–IDL bietet dagegen einen hierarchischen Namensraum, der durch geeignete Konventionen ebenfalls Name–Clashes verhindern kann und trotzdem einen inhaltlichen Bezug

zwischen Bezeichner und Schnittstelle zuläßt. Das CORBA–Äquivalent zu den GUIDs sind die Repository IDs.

Wie in CORBA werden in DCOM die Schnittstellen von Objekten mittels einer IDL beschrieben. In DCOM sind aber zusätzlich die Schnittstellen von Objekten auf Binärebene standardisiert, was die Grundlage für Programmiersprachenunabhängigkeit und Wiederverwendung von Kode in DCOM bildet.

Eine konzeptionelle Trennung von ORB und Object Services wie bei der OMA existiert in DCOM nicht. Demzufolge ist die Unterstützung einiger elementarer Systemdienste wie *Security* und *Lifecycle–Management* integraler Bestandteil von DCOM.

2.6 Überblick über CORBA–Implementierungen

Die Akzeptanz von CORBA läßt sich daran erkennen, daß in den letzten Jahren eine Reihe von CORBA–Implementierungen neu entstanden sind und einige proprietäre Middleware–Plattformen um CORBA–Funktionalität erweitert wurden.

Zu den bekannten kommerziellen Implementierungen zählen Orbix [22] von IONA, VisiBroker [66] von Visigenic, ORBplus [19] von HP, DSOM [20] von IBM und OmniBroker [43] von OOC. Daneben existieren einige frei verfügbare Implementierungen, wie omniORB [45] und JacORB [8]. Die eben genannten sind *allgemein* in dem Sinne, daß sie nicht auf ein bestimmtes Anwendungsgebiet spezialisiert sind. Als Ergebnis verschiedener Forschungsvorhaben sind außerdem CORBA–Implementierungen entstanden, die auf ein bestimmtes Anwendungsumfeld spezialisiert sind.

In [55] beschreiben Schmidt et al die Architektur von *The ACE ORB (TAO)*, einem Realtime Object Request Broker, der auf Basis der *Adaptive Communication Environment (ACE)* [54], einem Framework zur Entwicklung von verteilten Anwendungen, entwickelt wurde. Um Realtime–Fähigkeit zu erreichen, wurden eine Reihe von Änderungen bzw. Erweiterungen des CORBA–Standards vorgenommen, wie beispielsweise auf Realtime ausgelegte Objekt–Adapter, Inter–ORB–Protokolle und Scheduler.

ELEKTRA [30] konzentriert sich auf Ausfallsicherheit, indem Objekt–Implementierungen repliziert werden. Dafür werden spezielle Objekt–Adapter und Gruppenkommunikation eingeführt.

Alle genannten Produkte zeichnen sich dadurch aus, daß sie nicht oder nur in sehr geringem Umfang erweiterbar sind. Beispielsweise bietet Orbix sogenannte *Filter*, mit der Anwender nur an einigen wenigen Stellen in das CORBA–System eingreifen können.

Stellvertretend soll hier kurz auf das Design von OmniBroker [43] eingegangen werden, das typisch für die meisten der frei verfügbaren Implementierungen ist.

Im Gegensatz zu anderen kommerziellen Implementierungen ist der Quelltext von OmniBroker für die nicht kommerzielle Verwendung frei verfügbar.

Beim OmniBroker sind DII, SII, DSI, SSI, ORB–Kern, IIOP und BOA zu einem untrennbaren Ganzen verschmolzen, wobei die Funktionalität der einzelnen Komponenten über das ganze System verstreut ist. Beispielsweise befinden sich Teile der IIOP–Unterstützung in den vom IDL–Compiler generierten Stubs und Skeletten (Marshalling der Parameter nach IIOP–Konventionen) und im ORB–Kern (Verwaltung der IIOP–Kommunikationsendpunkte, IIOP–Protokollmaschine). Damit ist die Verwendung eines anderen Inter–ORB–Protokolls als IIOP nicht möglich, ohne Eingriffe am System vorzunehmen. Weiterhin ist die Funktionaliät von BOA und ORB–Kern nicht klar getrennt. So verwaltet der ORB–Kern eine Tabelle aller Objekte, was eigentlich Aufgabe des Objekt–Adapters wäre. Außerdem fehlt dem ORB–Kern ein Mechanismus, um zwischen mehreren Objekt–Adaptern auswählen zu können. Damit ist ein Austausch des BOA oder das Hinzufügen weiterer Objekt–Adapter nicht möglich.

Dieses Design läßt zwar einige Optimierungen in Hinsicht auf Performanz und Speicherbedarf zu, verhindert aber gleichzeitig wirksam die Erweiterbarkeit. Daher wurde in dieser Arbeit ein in den folgenden Kapiteln vorgestelltes konzeptionelles Rahmenwerk für eine *erweiterbare* CORBA–Implementierung entwickelt und implementiert.

Kapitel 3

Object Request Broker

Zentraler Bestandteil der CORBA–Architektur ist der *Object Request Broker (ORB)*; seine Hauptaufgabe ist die Weiterleitung von Methodenaufrufen vom Klienten zur Objekt–Implementierung einschließlich der damit in heterogenen Umgebungen zusammenhängenen Aufgabenstellungen.

Da der ORB eng mit den anderen Komponenten eines CORBA–Systems verbunden ist (teilweise über proprietäre Schnittstellen), waren die Entwickler von spezialisierten ORBs dazu gezwungen, auch die restlichen Komponenten des CORBA–Systems neu zu entwicklen bzw. nicht zur Verfügung zu stellen—wie bei einigen Forschungsprojekten zur Integration von *Quality of Service (QoS)* Aspekten in CORBA geschehen (ELEKTRA [30] und TAO [55]).

Hier liegt nun der Ansatzpunkt für MICO als Plattform für Forschungs– und Ausbildungsprojekte: die Enwicklung eines erweiter– und modifizierbaren ORB. Dazu wurde der Mikrokern–Ansatz aus dem Bereich der Betriebssysteme auf CORBA übertragen: Identifizierung der minimal notwendigen ORB–Funktionalität, so daß für Modifikationen und Erweiterungen in Frage kommende Teile als *Services* außerhalb des ORB–Kerns implementiert werden können.

Dazu werden in den nun folgenden Abschnitten zunächst die Aufgaben eines ORB zusammengestellt. Es wird analysiert, welche Komponenten zur Erfüllung dieser Aufgaben erforderlich sind, welche dieser Komponenten als ORB–Kern–Bestandteile und welche als Services realisiert werden. Schließlich wird ein Überblick über den Entwurf eines Mikrokern–ORBs gegeben.

3.1 Funktionalität

Der ORB stellt konzeptionell all die Funktionalität zur Verfügung, die unabhängig von bestimmten Objekttypen ist; jegliche objektspezifische Aufgaben werden durch Objekt–Adapter realisiert. Der ORB ist sozusagen der kleinste gemeinsame Nenner an Funktionalität unter allen denkbaren Arten von Objekten. Zu seinen Aufgaben zählt:

- Objektgenerierung
- Bootstrapping
- Methodenaufruf
  - Objektlokalisierung
  - Aufrufweiterleitung
  - Demultiplexen zwischen Objekt–Adaptern

Objektreferenzen enthalten Adressierungsinformationen, Typ und Identität eines Objektes. Die Generierung von Typ und Identität unterliegt den Objekt–Adaptern, der ORB steuert die Adressierungsinformationen bei. Daher müssen Objekt–Adapter und ORB bei der Erzeugung neuer Objektreferenzen eng zusammenarbeiten.

Nachdem Objekte erzeugt wurden, stellt sich die Frage, wie ein Klient Kenntnis von einer Objektreferenz erlangt. Dazu wird im einfachsten Fall ein Namensdienst verwendet. Da der Namensdienst selbst wieder als eine Menge von CORBA–Objekten realisiert ist, bleibt offen, wie der Klient die Objektreferenz des Namensdienstes erfährt. Zur Lösung dieses Bootstrapping–Problems bietet der ORB Klienten einen eingebauten Bootstrapping–Namensdienst an.

Ein Methodenaufruf beinhaltet aus Sicht des ORB drei Teilaufgaben: Nachdem ein Methodenaufruf durch einen Aufruf–Adapter initiiert wurde, muß erstens anhand der Objektreferenz des Zielobjektes dessen Aufenthaltsort festgestellt werden. Falls sich das Zielobjekt nicht im Adreßraum des Aufrufers befindet, muß zweitens der Methodenaufruf unter Verwendung geeigneter Transportmechanismen in den Adreßraum des Zielobjektes weitergeleitet werden. Dort wird drittens anhand der Objektreferenz des Zielobjektes der Objekt–Adapter bestimmt, der für das Zielobjekt zuständig ist. Dieser führt den Methodenaufruf auf dem Zielobjekt schließlich durch. Eventuelle Ergebnisse gelangen auf umgekehrtem Wege zurück zum Aufrufer.

3.2 Realisierungen

Konzeptionell gesehen ist der ORB ein wie in Abbildung 3.1 gezeigtes adreßraumübergreifendes System, das transparent die Kommunikation zwischen Objekten in den verschiedenen Adreßräumen erlaubt.

Man hat nun mehrere Möglichkeiten, diese konzeptionelle Vorstellung in eine Implementierung umzusetzen. Im wesentlichen unterscheidet man den zentralen und den dezentralen Ansatz (siehe Abbildung 3.2). Bei der dezentralen Lösung ist der ORB als Bibliothek realisiert, die zu jeder CORBA–Applikation gebunden wird. Die ORBs in verschiedenen Adreßräumen kommunizieren dann auf direktem Wege miteinander. Aus Sicherheits–, Zuverlässigkeits– und Performanz–Gründen

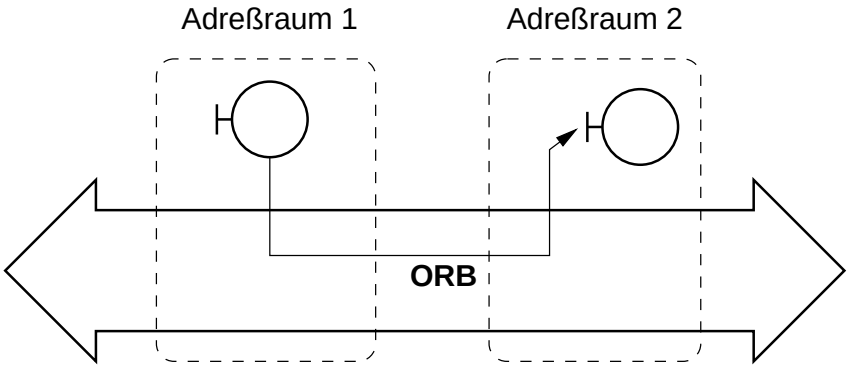


Abbildung 3.1: ORB als adreßraumübergreifender Objektbus

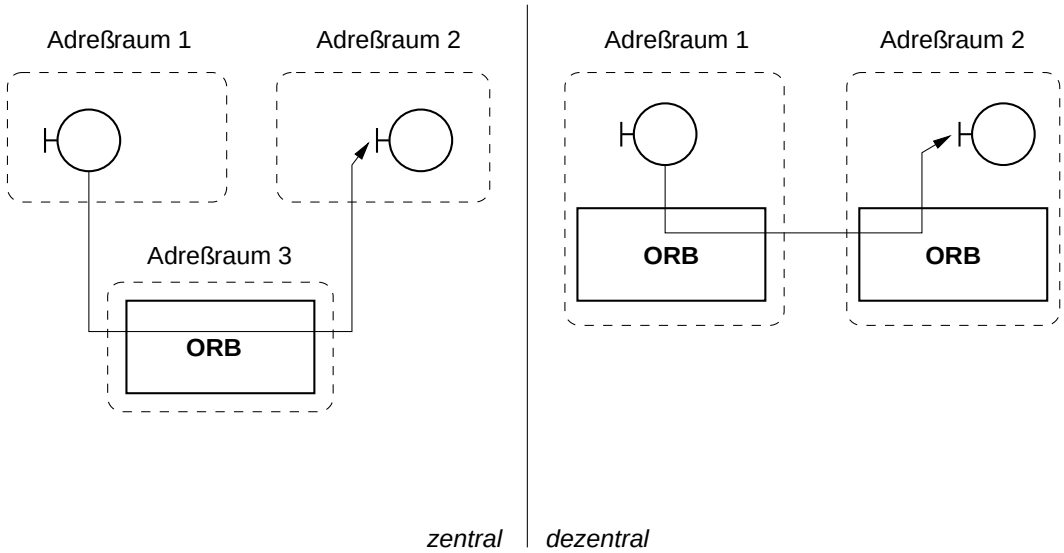


Abbildung 3.2: zentrale vs. dezentrale ORB–Realisierung

könnte sich diese Bibliothek auch direkt im darunterliegenden Betriebssystem befinden. Der dezentrale Ansatz weist folgende Eigenschaften auf:

+ gute Performanz, da Klienten und Server direkt miteinander kommunizieren

– aufwendiges Management, da Informationen über die lokalen ORBs verteilt sind

Bei der zentralen Lösung wird der ORB als separater Prozeß realisiert. Jegliche Kommunikation zwischen zwei CORBA–Applikationen läuft dann über diesen Server. Natürlich sind auch Mischlösungen denkbar, bei denen ein Teil der Kommunikation nicht den Umweg über den zentralen ORB nimmt. Eigenschaften des zentralen Ansatzes sind:

+ einfaches Management, da alle Informationen an einer zentralen Stelle vorliegen

– geringe Performanz, da Kommunikation zwischen Klienten und Servern nicht direkt erfolgt

– zentraler Ausfallpunkt

– schlechte Skalierbarkeit

Von Grund auf neu entwickelte CORBA–Implementierungen (wie auch MICO) verwenden wegen der mit der zentralen Lösung verbundenen Nachteile fast ausschließlich den dezentralen Ansatz. Der zentrale Ansatz kommt beispielsweise dann zur Anwendung, wenn neue Programmiersprachen oder –systeme an einen bestehenden ORB angebunden werden soll.

3.3 Design des MICO–ORB

Um maximale Flexibilität in Hinblick auf Erweiter– und Modifizierbarkeit zu erreichen, sollte es möglich sein, einen ORB um an spezielle Anforderungen angepaßte Aufruf–Adapter, Objekt–Adapter und Transportprotokolle zu erweitern. Dabei müssen beliebig viele dieser Komponenten parallel verwendbar sein. Für einen Mikrokern–ORB ergibt sich daraus, daß Aufruf–Adapter, Objekt–Adapter und Transportmodule als Services außerhalb des ORB angeordnet werden müssen und der ORB dafür geeignete Schnittstellen zur Verfügung stellen muß.

Aus Sicht des ORB lassen sich die genannten Komponenten in zwei Kategorien einteilen: Komponenten, die Methodenaufrufe auslösen und Komponenten, die Methodenaufrufe ausführen:

- Methodenaufruf–auslösende Komponenten:
  - Aufruf–Adapter (z.B. DII)
  - serverseitige Transportmodule (empfangen Netzwerkpaket und lösen beim lokalen ORB einen Methodenaufruf aus)
- Methodenaufruf–ausführende Komponenten:
  - Objekt–Adapter
  - klientenseitige Transportmodule (übernehmen Methodenaufruf vom lokalen ORB und verschicken Netzwerkpaket)

Es genügt also, den ORB mit einer verallgemeinerten *Aufruf–Adapter–Schnittstelle* sowie einer verallgemeinerten *Objekt–Adapter–Schnittstelle* auszurüsten. Neben diesen Schnittstellen verfügt der ORB über eine sogenannte

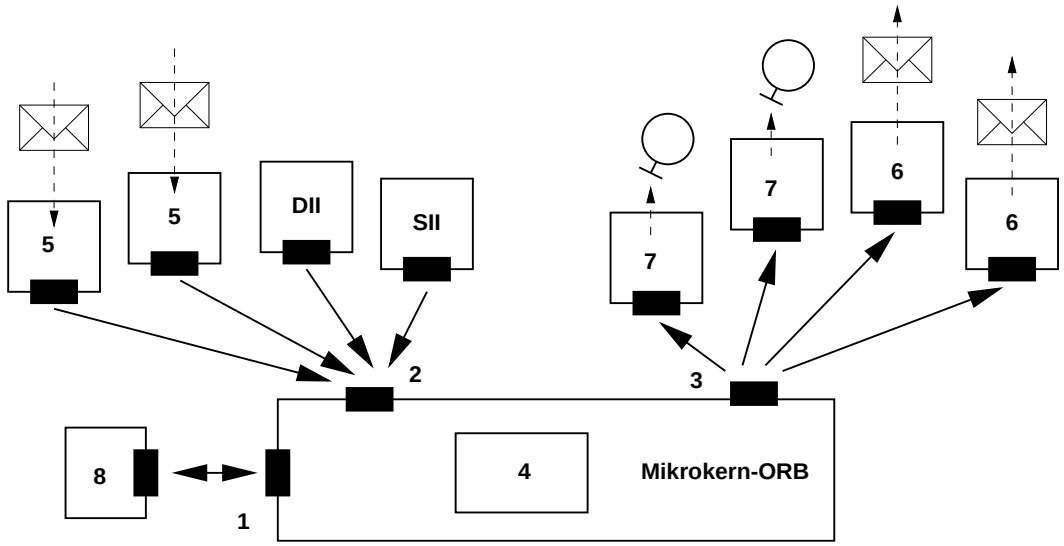


Abbildung 3.3: Mikrokern–ORB

Aufruftabelle, in der er Buch über gerade in Ausführung befindliche Methodenaufrufe führt.

Da in einem CORBA–System viele Teilaufgaben anfallen, die quasi parallel abgearbeitet werden müssen (siehe 3.3.4), verfügt der ORB außerdem über einen Scheduler. Weil auch an diesen Scheduler ganz unterschiedliche Anforderungen gestellt werden (multi–threaded vs. single–threaded, echtzeitfähig, etc.), muß der Scheduler selbst auch als Service außerhalb des ORB realisiert werden; der ORB stellt wiederum nur eine geeignete *Scheduler–Schnittstelle* zur Verfügung.

In Abschnitt 3.1 wurde bereits angesprochen, daß die Erzeugung von Objektreferenzen einer engen Kooperation zwischen Objekt–Adapter (steuert Typ und Identität bei) und ORB (steuert Adresse bei) bedarf. Durch die Anordnung der Transportmodule außerhalb des ORB verfügt nicht mehr der ORB sondern die Transportmodule über die Adressen, unter denen ein Objekt zu finden ist. Der ORB muß daher einen Mechanismus zur Kommunikation zwischen den Transportmodulen und den Objekt–Adaptern bereitstellen.

Alle Schnittstellen sind so ausgelegt, daß es jederzeit zur Laufzeit möglich ist, Komponenten beim ORB an– und abzumelden. Da der Mikrokern–ORB eine Schnittstelle zum Nachladen von Modulen anbietet, ist es sogar möglich, den ORB zur Laufzeit um neue Objekt–Adapter, Transportmodule usw. zu erweitern.

Abbildung 3.3 zeigt die Komponenten des Mikrokern–ORBs im Überblick. Nicht dargestellt ist der Bootstrapping–Namensdienst. In den folgenden Ab-

schnitten wird auf die einzelnen Komponenten bzw. Schnittstellen des ORB näher eingegangen.

3.3.1 Aufruf–Adapter–Schnittstelle

Über die sogenannte *Aufruf–Adapter–Schnittstelle* kann beim ORB ein entfernter Methodenaufruf initiiert werden. Stub–Objekte verwenden zum Durchführen von Methodenaufrufen indirekt über das SII diese Schnittstelle. Von CORBA–Applikationen mittels dem DII generierte Methodenaufrufe verwenden ebenfalls indirekt diese Schnittstelle. Serverseitige Transportmodule empfangen Nachrichten von ORBs in anderen Adreßräumen und transformieren diese in Aufrufe auf die Aufruf–Adapter–Schnittstelle des lokalen ORB.

Die Schwierigkeit beim Entwurf der Aufruf–Adapter–Schnittstelle liegt darin, daß die diese Schnittstelle benutzenden Komponenten über verschieden detaillierte Informationen über den durchzuführenden Methodenaufruf verfügen. Aufruf–Adaptern wie DII oder SII ist die Signatur der aufgerufenen Methode inklusive der Typen der Parameter bekannt, während das für serverseitige Transportmodule in der Regel nicht gilt. Ursache dafür ist, daß Inter–ORB–Protokolle wie zum Beispiel IIOP die Menge der zu übertragenden Informationen zu minimieren suchen und daher nur Objektreferenz, Methodenname und die Werte der Parameter, nicht jedoch deren Typen übermitteln, da das aufgerufene Objekt diese anhand des Methodennamens rekonstruieren kann. Das serverseitige Transportmodul könnte sich mittels des Interface Repository die Typinformationen theoretisch ebenfalls beschaffen, allerdings ist es erstens durchaus möglich, daß zu einer Methode keine Typinformationen im Repository vorhanden sind, und zweitens würde dies eine erhebliche Performanzeinbuße bedeuten, da für jeden Methodenaufruf ein eventuell nicht lokales Repository konsultiert werden müßte.

Als Folge dessen können serverseitige Transportmodule aus dem Byte–Strom der kodierten Parameter wegen mangelnder Typinformationen in der Regel die Werte der Parameter des Methodenaufrufes nicht rekonstruieren. Das ist erst zu einem viel späteren Zeitpunkt möglich, nämlich dann wenn der Methodenaufruf beim Zielobjekt angelangt ist, das wie oben beschrieben die Signatur der aufgerufenen Methode kennt. Gleichzeitig sollte es dann für das Zielobjekt transparent sein, ob der Methodenaufruf lokal von DII, SII oder von einem Transportmodul initiiert wurde.

Zur Lösung dieses Problems wird das Prinzip der *verzögerten Auswertung*[1] angewendet. Dabei wird die Auswertung von Ausdrücken einer Programmiersprache solange verzögert, bis der Wert des Ausdrucks tatsächlich benötigt wird und alle für die Auswertung nötigen Eingabewerte vorliegen. Dazu wird für jeden

[1] Verzögerte Auswertung war in Algol 60 als Methode der Parameterübergabe *Namensaufruf* enthalten. Als weitere wichtige Anwendung dieser Technik ist die verzögerte Auswertung von Datenströmen in Lisp zu nennen, die zuerst in [28] beschrieben wurde.

Ausdruck, der verzögert ausgewertet werden soll, ein sogenanntes *Thunk–Objekt*[2] erzeugt, das den Ausdruck selbst und alle zu seiner Auswertung nötigen Informationen enthält. Erst wenn der Wert des Ausdrucks benötigt wird, wird der Thunk ausgewertet.

Für die Aufruf–Adapter–Schnittstelle des ORB bedeutet das, daß ein spezielles Thunk–Objekt für den Methodenaufruf generiert wird, das die kodierten Parameter und weitere zum Dekodieren notwendige Informationen (wie etwa das zum Dekodieren zu verwendende Verfahren) enthält. Erst wenn die Werte der Parameter zum Aufruf der Methode tatsächlich benötigt werden, wird das Thunk–Objekt mit den dann bekannten Typinformationen versorgt und ausgewertet (d.h., die Parameter werden dekodiert).

Um mehrere Methodenaufrufe parallel absetzen zu können, muß die Aufruf–Adapter–Schnittstelle asynchron sein, das heißt der Kontrollfluß kehrt nach Absetzen eines Methodenaufrufs beim ORB direkt zum Aufrufer zurück. Die Semantik des unter CORBA–üblichen synchronen Methodenaufrufes kann damit simuliert werden. Die Schnittstelle unterstützt die Operationen:

$$\begin{aligned} \text{invoke}(O, M, T) &\rightarrow H \\ \text{cancel}(H) &\rightarrow \{\} \\ \text{wait}(\{H_i\}) &\rightarrow H \\ \text{results}(H) &\rightarrow T \end{aligned}$$

invoke übergibt dem ORB einen Methodenaufruf in Form eines Tupels (O, M, T) bestehend aus Zielobjekt O, Methodenname M und Parametern in Form eines Thunks T und liefert ein Handle H für den initiierten Methodenaufruf zurück. Mittels *cancel* kann der zum angegebenen Handle gehörige Methodenaufruf abgebrochen werden. *wait* wartet auf das Beenden eines der in Form einer Menge von Handles angegebenen Methodenaufrufe und liefert das Handle des ersten beendeten Methodenaufrufes als Ergebnis. Schließlich kann man mit *results* die Ergebnisse des Methodenaufrufs in Form eines Thunks T beim ORB abholen. Auch für die Ergebnisse muß von der verzögerten Auswertung Gebrauch gemacht werden, da die Typen der Ausgabeparameter erst im Stub–Objekt bekannt sind, das den Methodenaufruf ausgelöst hat.

3.3.2 Objekt–Adapter–Schnittstelle

Die Objekt–Adapter–Schnittstelle ist gewissermaßen das Gegenstück zur Aufruf–Adapter–Schnittstelle. Der ORB benutzt sie, um einen Methodenaufruf an die für

[2]Der Name *Thunk* stammt von der Implementierung des Namensaufrufs in Algol 60. Der Ursprung des Wortes ist unbekannt, es wird aber erzählt, daß er dem Geräusch entspricht, das Daten in einem laufenden Algol 60 System von sich geben, wenn sie in den Keller geschrieben werden [1].

dessen Ausführung zuständige Komponente zu übergeben. Das kann entweder ein Objekt–Adapter sein, der den Methodenaufruf direkt durchführt, oder aber ein Transportmodul, das den Methodenaufruf in einen anderen Adreßraum weiterleitet.

Die Schwierigkeit beim Entwurf dieser Schnittstelle liegt darin, daß die Schnittstelle die Verschiedenartigkeit von Objekt–Adaptern und Transportmodulen vor dem ORB verbergen soll, es dem ORB aber trotzdem noch möglich sein muß, anhand der Objektreferenz des Zielobjektes zu entscheiden, welcher Objekt–Adapter bzw. welches Transportmodul für die Ausführung eines Methodenaufrufes auf dem Zielobjekt zuständig ist.

Die Lösung dieses Problems liegt darin, nicht den ORB, sondern die Objekt–Adapter bzw. Transportmodule diese Auswahl treffen zu lassen. Die Schnittstelle, die von den Objekt–Adaptern bzw. Transportmodulen dem ORB angeboten wird, umfaßt folgende Operationen:

$$\begin{aligned} \text{has\_object}(O) &\rightarrow \{\text{TRUE}, \text{FALSE}\} \\ \text{invoke}(O, M, T, H) &\rightarrow \{\} \\ \text{cancel}(H) &\rightarrow \{\} \end{aligned}$$

Mittels *has_object* kann der ORB bei einem Objekt–Adapter bzw. Transportmodul nachfragen, ob er für die Objektreferenz O zuständig ist oder nicht. Eine wichtige Forderung an *has_object* ist, daß für jede mögliche Objektferenenz O *has_object*(O) von höchstens einem Objekt–Adapter bzw. Transportmodul wahr wird. Denn nur so ist für den ORB eine eindeutige Zuordnung von Objektreferenzen möglich. *invoke* übergibt einen Methodenaufruf in Form eines Tupels (O, M, T, H) bestehend aus Objektreferenz O, Methodenname M, Parametern in Form eines Thunks T und einem Handle H dem Objekt–Adapter bzw. Transportmodul. Mit *cancel* kann der ORB einen Methodenaufruf gegebenenfalls abbrechen.

Um mehrere Methodenaufrufe parallel absetzen zu können, muß auch die Objekt–Adapter–Schnittstelle asynchron sein. Daher verfügt die den Objekt–Adaptern bzw. Transportmodulen vom ORB angebotene Schnittstelle neben Operationen zum An– und Abmelden von Objekt–Adaptern über eine Operation *answer_invoke*.

$$\begin{aligned} \text{register}(OA) &\rightarrow \{\} \\ \text{unregister}(OA) &\rightarrow \{\} \\ \text{answer\_invoke}(H, T) &\rightarrow \{\} \end{aligned}$$

Mittels *answer_invoke* gibt ein Objekt–Adapter bzw. Transportmodul dem ORB die Ergebnisse des durch das Handle H spezifizierten Methodenaufrufes in Form eines Thunks T bekannt.

3.3.3 Aufruftabelle

Aufgrund der Asynchronität der Objekt–Adapter–Schnittstelle muß der ORB Buch über die gerade aktiven Methodenaufrufe führen. Zu diesem Zweck verfügt der ORB über eine Tabelle der gerade aktiven Methodenaufrufe, die Einträge der Form $(H, O, M, T_{in}, T_{out})$, bestehend aus Handle H, Objektreferenz O des Zielobjektes, Methodenname M, Thunk T_{in} für Eingabeparameter und (einem eventuell leeren) Thunk T_{out} für Ergebnisse, enthält.

Ein Methodenaufruf läuft damit aus Sicht des ORB in drei Schritten ab. Im ersten Schritt nimmt der ORB einen Methodenaufruf von einer aufrufgenerierenden Komponente entgegen und leitet ihn an die zuständige aufrufausführende Komponente weiter:

1. Entgegennahme eines Methodenausrufes (O, M, T_{in}) an der Aufruf–Adapter–Schnittstelle mittels *invoke*
2. Generierung eines neuen Handles H und Eintragung von $(H, O, M, T_{in}, \text{NIL})$ in die Aufruftabelle
3. Auswahl der zuständigen aufrufausführenden Komponente und Übergabe des Methodenaufrufes an diese
4. H wird als Ergebnis von *invoke* zurückgeliefert

Daraufhin bearbeitet die aufrufausführende Komponente den Methodenaufruf und teilt dem ORB schließlich die Ergebnisse des Methodenaufrufs mit:

5. Entgegennahme der Ergebnisse T_{out} für den Methodenaufruf mit Handle H an der Objekt–Adapter–Schnittstelle mittels *answer_invoke*
6. Ersetzung des zugehörigen Eintrag $(H, O, M, T_{in}, \text{NIL})$ in der Tabelle der aktiven Methodenaufrufe durch $(H, O, M, T_{in}, T_{out})$

Im dritten und letzten Schritt holt sich die aufrufgenerierende Komponente mittels $results(H)$ die Ergebnisse des Methodenaufrufes mit Handle H an der Aufruf–Adapter–Schnittstelle ab:

7. Finde Eintrag mit Handle H in der Tabelle der Aufruftabelle
8. T_{out} wird als Ergebnis von *invoke* zurückgeliefert und der Eintrag mit Handle H wird aus der Aufruftabelle entfernt

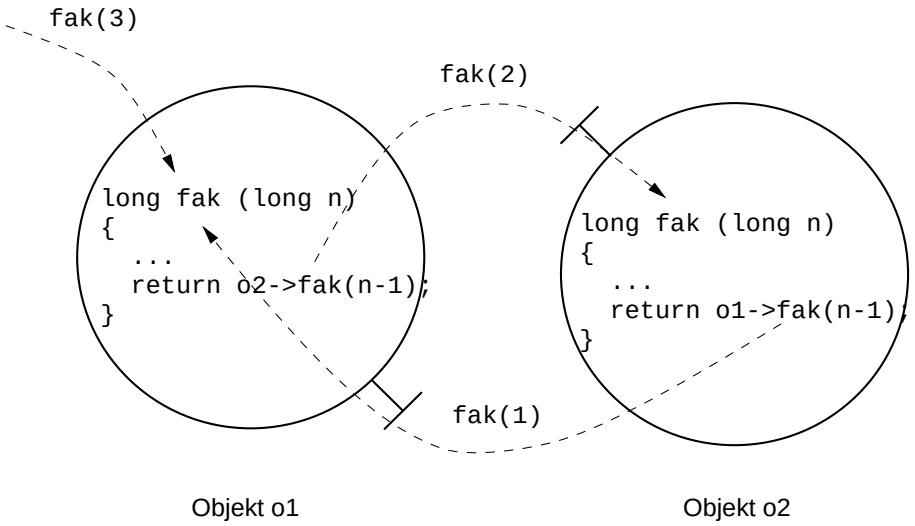


Abbildung 3.4: Verteilte Berechnung von $n!$

3.3.4 Scheduler

In einem komplexen System wie dem ORB gibt es viele Situationen, in denen mehrere Aufgaben quasi gleichzeitig ausgeführt werden müssen. Die folgenden Beispiele zeigen einige dieser Situationen.

Beispiel 1: Eine Realtime–Erweiterung des Mikrokern–ORB implementiert zwei Transportprotokolle, das auf TCP basierende IIOP und ein spezielles auf ATM basierendes *Realtime Inter–ORB Protocol (RIOP)*. Unabhängig voneinander müssen beide Transportmodule auf eintreffende TCP bzw. ATM Nachrichten warten und diese auswerten.

□

Beispiel 2: Es gibt Situationen, in denen zwei oder mehrere CORBA–Objekte wechselseitig Methoden aufeinander aufrufen. Ein Beispiel ist eine verteilte Applikation zur Berechnung von $n!$. In zwei verschiedenen Prozessen befindet sich je ein CORBA–Objekt, das über eine Methode `fak(n)` zur rekursiven Berechnung der Fakultät von `n` verfügt. Im allgemeinen Fall berechnet diese Methode `fak(n-1)` und multipliziert das Ergebnis mit `n`. Dazu ruft `fak()` jedoch nicht sich selbst, sondern die Methode des jeweils *anderen* Objektes auf. Dadurch entsteht die in Abbildung 3.4 angedeutete geschachtelte Folge von wechselseitigen Methodenaufrufen[3] bei der Berechnung der Fakultät von 3. Damit das funktioniert und nicht zu einem Deadlock führt, muß der ORB in der Lage sein, gleichzeitig auf die Beendigung des Methodenaufrufes und das Eintreffen von neuen Methodenaufrufen zu warten.

□

[3] auch bekannt als *nested method invocations*

Erschwerend kommt hinzu, daß für den ORB nicht alle diese Situationen vorhersehbar sind, da beliebige Transportmodule zur Laufzeit ins System integriert und wieder entfernt werden können. Als erweiterbare Plattform sollte der Mikrokern–ORB sowohl in single– wie auch multi–threaded Umgebungen funktionieren. Das Problem kann also nicht einfach durch Einsatz einer Thread–Package gelöst werden. Folglich muß der ORB selbst irgendeine Form von Scheduler anbieten, der die Rechenzeit auf die gleichzeitig auszuführenden Programmteile verteilt. Da die Scheduling–Algorithmen aber hochgradig vom Einsatzgebiet des ORBs abhängen (man denke beispielsweise an eine Realtime–Umgebung), sollte der Scheduler austauschbar sein und muß daher als Service außerhalb des Mikrokern–ORBs angesiedelt sein.

Scheduler–Abstraktion

Gesucht ist ein Scheduling–Mechanismus mit folgenden Eigenschaften:

- funktioniert in single– und multi–threaded Umgebungen
- Trennung von Schnittstelle und Implementierung
- erlaubt die Implementierung verschiedener Scheduling–Algorithmen

Zum Entwurf eines solchen Mechanismus werden folgende Annahmen über die Teilaufgaben gemacht:

1. Der Kode der parallel auszuführenden Teilaufgaben läßt sich in zwei Kategorien einteilen:
   - *Warteoperationen*, die auf das Eintreten eines bestimmten Ereignisses warten
   - *Rechenoperationen* sind alle Operationen, die keine Warteoperationen sind
2. Die Zeit zur Durchführung jeder Folge von Rechenoperationen einer Teilaufgabe ist vernachlässigbar klein
3. Es gibt nur eine begrenzte Zahl verschiedener Ereignisse

Zur Annahme 1 ist folgendes zu bemerken: Es gibt einige Operationen, die sich auf den ersten Blick nicht eindeutig in eine der beiden Kategorien einordnen lassen, sondern eine Kombination aus beiden sind. Beispielsweise können Ein/Ausgabe–Operationen unter Umständen solange blockieren, bis die Ein– bzw. Ausgabe abgeschlossen ist. Jedoch kann man solche Misch–Operationen in eine Menge von reinen Warte– und Rechenoperationen transformieren.

Beispiel 3: Die Operation

```
lese_blockierend (socket, puffer, puffer_groesse)
```

liest von einer Netzwerkverbindung, die durch `socket` repräsentiert wird, die in `puffer_groesse` angegebene Anzahl von Bytes in `puffer`. Die Operation blockiert solange, bis die in `puffer_groesse` angegebene Anzahl Bytes empfangen wurden. `lese_blockierend()` ist also weder Warte– noch Rechenoperation, sondern beides. Das Kodestück

```
while (puffer_groesse > 0) {
  warte_auf_daten (socket);
  gelesen = lese_nicht_blockierend (socket, puffer,
                                    puffer_groesse)
  puffer += gelesen;
  puffer_groesse -= gelesen;
}
```

liest ebenfalls von der Netzwerkverbindung `socket` die in `puffer_groesse` angegebene Anzahl von Bytes in `puffer`. Jedoch wird dazu die Warteoperation `warte_auf_daten(socket)`, die auf das Ereignis *Auf der Verbindung* `socket` *sind Daten angekommen* wartet, und die Rechenoperation `lese_nicht_blockierend` verwendet, die ohne zu blockieren liest, was im Moment gerade verfügbar ist.

□

Zu Annahme 2: Mit „vernachlässigbar“ ist hier gemeint, daß für jede beliebig aber fest vorgegebene Zeitspanne $\Delta t > 0$ die Zeit zur Ausführung jeder Folge von Rechenoperationen aller Teilaufgaben kleiner oder gleich Δt ist. Das diese Annahme gemacht werden kann, ist keineswegs offensichtlich. Man beachte jedoch, daß sich jede Folge von Rechenoperationen, die dieser Bedingung nicht genügt, in eine Menge von äquivalenten Rechenoperationen transformieren läßt, die alle die Bedingung erfüllen. Dazu fügt man einfach in der (zeitlichen) Mitte einer solchen Folge eine Dummy–Warteoperation ein. Daraus resultieren zwei neue Folgen von Rechenoperationen, die entweder der Bedingung genügen oder nicht. Falls nicht, so wiederholt man den Vorgang solange, bis alle Folgen die Bedingung erfüllen.

Zu Annahme 3: Im Prinzip genügt ein Primitiv zur Interprozeßkommunikation wie beispielsweise die Semaphore mit dem zugehörigen Ereignis *Down auf Semaphore X möglich*. Alle denkbaren Ereignisse können auf diese Primitive abgebildet werden [61]. Jedoch treten einige Ereignisse so häufig auf, daß es sinnvoll ist, sie direkt zur Verfügung zu stellen:

- Eingabe auf Kanal X möglich
- Ausgabe auf Kanal X möglich
- Zeitspanne T verstrichen

Wie oben dargestellt enthält ein CORBA–System verschiedene Teilaufgaben, die quasi parallel ausgeführt werden müssen. Mit Annahme 1 läßt sich der Programmkode jeder dieser Teilaufgaben in Blöcke zerlegen, an deren Anfang eine Warteoperation steht und deren Rest aus Rechenoperationen besteht. Da jede Teilaufgabe für sich seriell ausgeführt wird, ist von jeder Teilaufgabe immer genau ein Block aktiv (d.h. wird ausgeführt). Da mit Annahme 2 jedoch die Zeit zur Ausführung der Rechenoperationen vernachlässigbar ist, wird praktisch zu jeder Zeit auf genau ein Ereignis pro Teilaufgabe gewartet. Das Scheduling zwischen den Teilaufgaben beschränkt sich nun darauf, auf das Eintreten des nächsten dieser Ereignisse zu warten und den zugehörigen Block auszuführen. Da mit Annahme 2 die Rechenzeit dafür vernachlässigbar ist, kann in vielen Fällen auf *Preemption* verzichtet werden, d.h. die Blöcke werden ohne Unterbrechung bis zu Ende ausgeführt. In Echtzeit–Umgebungen, wo Teilaufgaben bis zum Ablauf einer bestimmten *Deadline* abgearbeitet sein müssen, ist das natürlich nicht möglich. Die hier vorgestellte Scheduler–Abstraktion erlaubt aber auch den Einsatz von Scheduling–Algorithmen, die Preemption unterstützen, beispielsweise auf Basis einer Thread–Package.

Die Idee ist nun, das Warten auf die Ereignisse in den Scheduler zu verlagern, da nach Annahme 3 die Zahl verschiedener Ereignisse begrenzt ist. Beim „Betreten“ eines Blockes wird das zugehörige Ereignis beim Scheduler registriert, wobei sich der Scheduler den zum Ereignis gehörigen Block merkt. Mittels geeigneter Mechanismen wartet der Scheduler nun auf das Eintreten des nächsten registrierten Ereignisses. Der Scheduler entfernt daraufhin den entsprechenden Eintrag aus seiner Liste und führt den zum Ereignis gehörigen Block aus. Wie beim gleichzeitigen Eintreten mehrerer Ereignisse verfahren wird, regelt der verwendete Scheduling–Algorithmus.

Beispiel 4: Abbildung 3.5 zeigt zwei Teilaufgaben A und B, die in zwei bzw. einen Block zerlegt wurden. Jeweils am Anfang eines Blocks steht eine Warteoperation, der schattierte Rest besteht aus Rechenoperationen. Die gestrichelten Pfeile spiegeln den Kontrollfluß zwischen den Blöcken wider. Die Ausführung der beiden Teilaufgaben beginnt bei Block A_1 bzw. B_1, also werden initial die zugehörigen Ereignisse E_1 und E_3 beim Scheduler registriert. Der Scheduler verfügt über eine Methode `schedule()`, die das Scheduling implementiert. Diese wird zu Beginn aufgerufen und bis zum Programmende nicht mehr verlassen. Der Scheduler wartet nun darauf, daß Ereignis E_1 oder E_3 eintritt. Angenommen E_1 tritt zuerst ein, so wird E_1 aus der Liste des Schedulers entfernt und Block A_1 ausgeführt. Gemäß des Kontrollflußes muß Block A_2 als nächstes ausgeführt werden, also wird das Ereignis E_2 beim Scheduler registriert und das Scheduling beginnt von vorn.

□

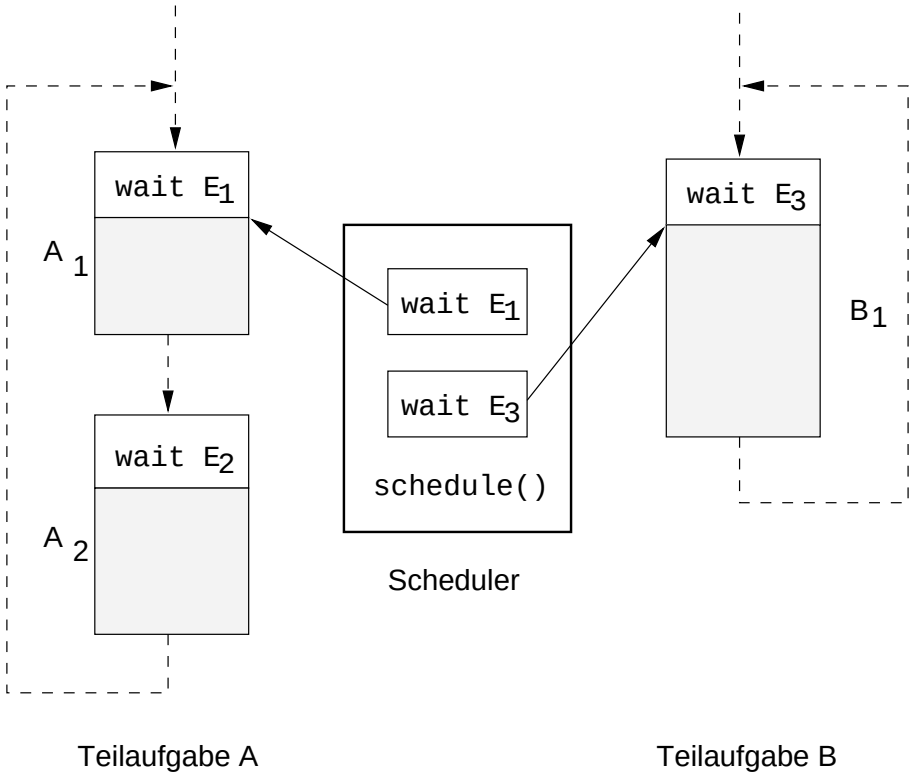


Abbildung 3.5: Scheduler

Eigenschaften dieses Ansatzes sind:

+ vollständige Trennung von Schnittstelle und Implementierung des Schedulers

+ funktioniert in single–threaded Umgebungen und kann die Vorteile einer multi–threaded Umgebung nutzen (beispielsweise ein Thread pro Teilaufgabe)

– Kode muß speziell strukturiert werden

– keine Preemption in single–threaded Umgebungen

Scheduler–Schnittstelle

Der Entwurf einer allgemeinen Schnittstelle zu der im letzten Abschnitt vorgestellten Scheduler–Abstraktion ist deswegen schwierig, weil jeder Scheduling–Algorithmus spezielle Konfigurationsdaten benötigt. *Round–Robin* benötigt beispielsweise die Dauer einer Zeitscheibe, beim *Scheduling nach Prioritäten* muß für jede Teilaufgabe eine Priorität vergeben werden. Der Anwender muß sich also

des verwendeten Scheduling–Algorithmus bewußt sein, was eine Kapselung aller Scheduling–Algorithmen durch eine allgemeingültige Schnittstelle verhindert.

Andererseits hat es sich gezeigt, daß in einem CORBA–System die Verwendung eines speziellen Schedulers oftmals an die Verwendung spezieller weiterer Komponenten gebunden ist, die die Dienste des Schedulers in Anspruch nehmen. Für die Verwendung in Realtime–Umgebungen enthält TAO [55] beispielsweise spezielle Realtime–Inter–ORB–Protokolle und Realtime–Objekt–Adapter, die von einem Realtime–Scheduler Gebrauch machen. Um MICO mit Realtime–Eigenschaften auszustatten, müßten also gleichzeitig Scheduler *und* die Komponenten, die den Scheduler benutzen, ausgetauscht werden. Dadurch sind sich aber die den Scheduler benutzenden Komponenten des im ebenfalls ausgetauschten Scheduler verwendeten Algorithmus bewußt.

MICO–Scheduler bieten daher zwei Schnittstellen, eine allgemeine von Konfigurationsdaten spezieller Algorithmen (wie beispielsweise Prioritäten, Deadlines, etc.) unabhängige und eine speziell auf den verwendeten Algorithmus zugeschnittene. Bei Aufrufen auf der allgemeinen Schnittstelle werden die fehlenden Konfigurationsdaten durch geeignete Default–Werte ersetzt. Gemeinsam mit dem Scheduler „nachgerüstete“ Komponenten verwenden die spezielle Schnittstelle, während die restlichen Komponenten, die kein Wissen über den verwendeten Scheduling–Algorithmus haben, die allgemeine Schnittstelle verwenden.

3.3.5 Objektgenerierung

Objektreferenzen sind ein Tupel $(\{L_i\}, T, I)$ bestehend aus Lokatoren L_i, Typ T und Identität I eines Objektes. Eine Lokator beschreibt dabei einen Mechanismus, wie auf das Objekt zugegriffen werden kann[4]. Da es durchaus verschiedene Mechanismen zum Zugriff auf ein Objekt geben kann, verfügt jede Objektreferenz nicht nur über einen, sondern eine Menge von Lokatoren.

Beispiel: Die eben beschriebenen Lokatoren ähneln in ihrer Funktion den aus dem WWW–Umfeld bekannten *Uniform Resource Locators (URL)* [6]. Naheliegenderweise ähnelt die vom Benutzer sichtbare Syntax von Lokatoren in MICO der von URLs. Lokatoren für Objekte, auf die mittels IIOP zugegriffen werden kann, folgen dem Format

```
iiop:<ip-addresse>:<port-nummer>
```

[4] Jeder Lokator ist auch eine Adresse, jedoch nicht umgekehrt. Der Unterschied zwischen beiden ist die eher abstrakte Natur der Adresse, die zwar ein Objekt eindeutig identifiziert, jedoch nicht notwendigerweise einen Weg zum Auffinden des Objektes beschreibt. Beispielsweise ist das Tupel (Land, Stadt, Straße, Hausnummer) eine Adresse, aber kein Lokator. Ein Lokator wäre beispielsweise eine Wegbeschreibung.

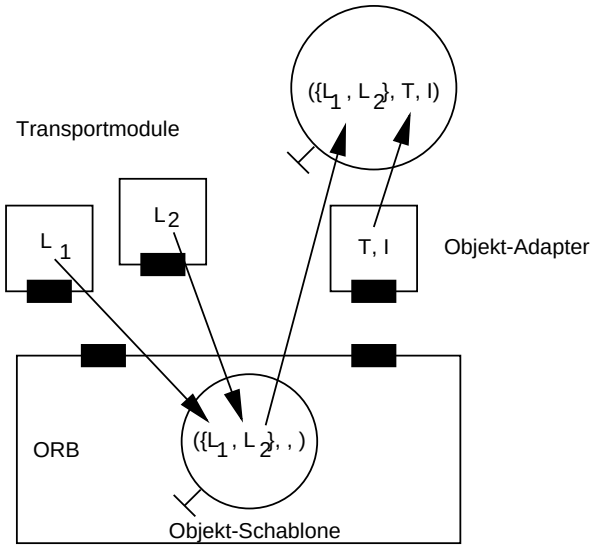


Abbildung 3.6: Generierung von Objektreferenzen

Daneben wird die Kommunikation zwischen Prozessen auf ein und demselben Rechner mittels sogenannter *Named Pipes* unterstützt, die im Dateisystem als spezielle Dateien sichtbar werden. Lokatoren dafür haben das Format

```
pipe:<pfad>
```

Der spezielle Lokator

```
local:
```

dient zum Zugriff auf Objekte im selben Adreßraum.

□

Bei der Erzeugung neuer Objektreferenzen arbeiten Objekt–Adapter eng mit dem ORB zusammen: der Objekt–Adapter steuert Typ und Identität bei, die Lokatoren stammen von den serverseitigen Transportmodulen, die den Zugriff auf den Adreßraum von außen ermöglichen. Zur Kommunikation zwischen Objekt–Adaptern und Transportmodulen stellt der ORB wie in Abbildung 3.6 dargestellt eine Schablone bereit, in der die Transportmodule Lokatoren eintragen. Durch Ergänzung dieser Schablone um Typ und Identität generiert ein Objekt–Adapter eine neue Objektreferenz.

3.3.6 Bootstrapping

Eine elementare Aufgabe in einem verteilten System ist die *Dienstvermittlung*, d.h. die Zusammenführung von Dienstanbietern und Dienstnutzern. Im wesentlichen unterscheidet man zwei verschiedene Ansätze, die *namensorientierte* und die

inhaltsorientierte Vermittlung. Im ersten Fall erfolgt die Vermittlung über Namen, die der Anwender an Dienste vergibt. Im zweiten Fall erfolgt die Vermittlung über Eigenschaften des Dienstes. Bei Vermittlern des ersten Typs spricht man von *Namensdiensten*, beim zweiten Typ von *Tradern*.

In CORBA sind diese Vermittler als Object Service, das heißt als Menge von CORBA–Objekten außerhalb des ORB–Kerns, realisiert. Dadurch ergibt sich jedoch die Frage, wie ein Dienstnutzer Kenntnis von der Objektreferenz des Vermittlers (Namensdienst oder Trader) selbst erhält, was auch als *Bootstrapping–Problem*[5] bezeichnet wird.

Die CORBA–Spezifikation bietet eine Lösung für dieses Problem, indem der ORB eine Schnittstelle zum Abfragen von sogenannten *initialen Objekreferenzen* anhand eines Namens (wie beispielsweise `NameService` für die Objektreferenz des Namensdienstes) anbietet:

```
// IDL
interface ORB {
  ...
  Object resolve_initial_references (in string name);
  ...
};
```

Eine Realisierung dieser Schnittstelle durch einen ORB erfordert selbst wieder einen Namensdienst mit folgenden Eigenschaften:

- nur lesender Zugriff
- verteilte Datenbasis

Mit *verteilter Datenbasis* ist gemeint, daß sich Informationen über die zu vermittelnden Objekte in verschiedenen Adreßräumen befinden können. Beispielsweise ist es denkbar, daß in zwei verschiedenen Adreßräumen Namensdienste zur Verfügung stehen. Daher definiert der Bootstrapping–Namensdienst eine Abbildung

$$(L, T, I) \rightarrow O$$

zwischen Namen der Form (L, T, I) und Objektreferenzen O. Eine Name besteht aus einem Lokator L, der den zu durchsuchenden Adreßraum identifiziert, dem Typ T des gesuchten Objektes und einem Identifizierer I, der benötigt wird, um Objekte mit gleichem Typ unterscheiden zu können.

Ein solcher Bootstrapping–Namensdienst kann nicht mittels des normalen CORBA–Aufrufmechanismus implementiert werden, da initial keine Objektreferenz vorhanden ist, auf der Methodenaufrufe durchgeführt werden könnten. Es

[5] Umgangssprachlich auch als oder *Henne–Ei–Problem* bekannt.

bleiben daher zwei Möglichkeiten zur Realisierung: Erstens kann ein CORBA–fremder Mechanismus verwendet werden. Nachteilig dabei ist, daß stets ein externer Dienst aktiv sein muß und sich die zu vermittelnden Objekte dort registrieren müssen. Diese Nachteile entfallen beim zweiten Ansatz, bei dem der Bootstrapping–Namensdienst zum integralen Bestandteil des ORB gemacht wird, weshalb dieser Ansatz auch für MICO gewählt wurde.

3.3.7 Dynamische Erweiterbarkeit

Interpretierte Sprachen bieten die Möglichkeit, ein momentan ausgeführtes Programm jederzeit zu erweitern, indem neue Programmstücke in Form von Quelltext geladen und ausgeführt (interpretiert) werden.

Bei nichtinterpretierten Programmiersprachen wie C++ wird dies dadurch erschwert, daß vor dem Ausführen eines Programms ein *Link*–Vorgang nötig ist, der Querverweise zwischen den einzelnen Programmodulen auflöst. Moderne Betriebssysteme stellen jedoch einen sogenannten *dynamischen Linker* zur Verfügung, der diese Querverweise auch in einem bereits ausgeführten Programm auflösen kann, wenn ein neues Modul hinzugefügt werden soll. Mittels eines dynamischen Linkers werden somit die Dynamik von interpretierten und die Effizienz von übersetzten Sprachen kombiniert.

Durch Verwendung des dynamischen Linkens bietet MICO die Möglichkeit, ein laufendes CORBA–System um neue Module, wie beispielsweise Objekt–Adapter, zu erweitern.

3.4 Zusammenfassung, Bewertung und Alternativen

Zum Entwurf einer erweiter– und modifizierbaren CORBA–Plattform wurde der Mikrokern–Ansatz auf CORBA übertragen. Dafür wurden die Komponenten eines CORBA–Systems in Mikrokern–Bestandteile und Services (Komponenten außerhalb des Mikrokern–ORB) eingeteilt und es wurden die nötigen Schnittstellen des Mikrokern–ORB entworfen. Dabei wurde festgestellt, daß sich die Services im wesentlichen in *aufrufgenerierende* und *aufrufausführende* einteilen lassen, die je eine spezielle Schnittstelle zum Mikrokern–ORB erfordern. Wie in Abbildung 3.7 gezeigt, besteht dadurch die Funktion des Mikrokern–ORB bei einem Methodenaufruf vereinfacht gesagt in der Selektion einer aufrufausführenden Komponente.

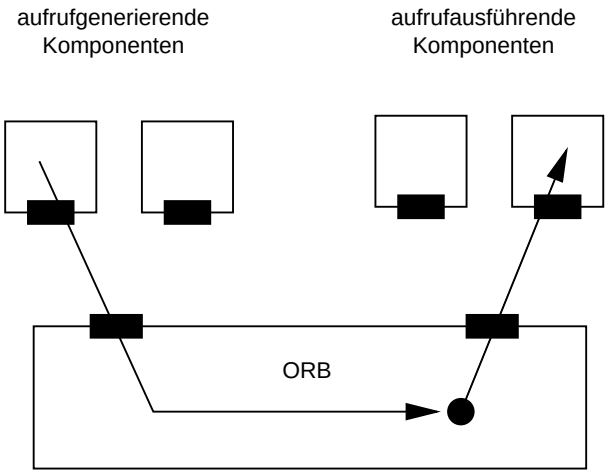


Abbildung 3.7: Mikrokern–ORB als Selektor beim Methodenaufruf

Der Mikrokern–ORB weist folgende Eigenschaften auf:

+ einfache Erweiterbarbeit durch Hinzufügen bzw. Austauschen von Services

– Erweiterbarkeit ist auf Service–Arten beschränkt, die beim Entwurf des Mikrokern–ORBs berücksichtigt wurden

– erhöhter Verwaltungsaufwand

Der erhöhte Verwaltungsaufwand ist ein Nachteil, den Generizität oftmals mit sich bringt. Herkömmliche CORBA–Systeme, bei denen ein bestimmter Kommunikationsmechanismus, ein bestimmter Objekt–Adapter und der ORB–Kern zu einer großen monolithischen Komponente verschmelzen, lassen eine Vielzahl von Optimierungen zu, die durch die Trennung der Komponenten bei Verwendung eines Mikrokern–ORB nicht möglich sind. Beispielsweise verfügen einige herkömmliche CORBA–Systeme über eine zentrale Tabelle, die CORBA–Objekten Servants (siehe Kapitel 5) und Kommunikationsendpunkte (siehe Kapitel 4) zuordnet, während beim Mikrokern–ORB jedes Kommunikationsmodul und jeder Objekt–Adapter über separate Tabellen verfügen muß.

Die Beschränkung der Erweiterbarkeit auf Service–Arten, die beim Entwurf des Mikrokern–ORB vorgesehen wurden, wird dann zum Problem, wenn der ORB um neue Kommunikations–Mechanismen erweitert werden soll. Beispielsweise wird momentan in der OMG diskutiert [37], wie isochrone Datenströme vom ORB unterstützt werden können. Da sich Datenströme nicht auf die Methodenaufrufsemantik der aktuellen CORBA–Spezifikation abbilden lassen, wird die Integration von Datenströmen unter Umständen Erweiterungen erfordern, die nicht als Service realisiert werden können und damit Änderungen am Mikrokern–ORB erfordern.

Ein anderer Ansatz zur Realisierung eines erweiterbaren ORB ist die Bereitstellung eines *Frameworks* [13] wie ACE [54] oder dem im ReTINA Umfeld entstehenden Framework [64], mit dessen Hilfe ORBs realisiert werden können. Die Idee dabei ist, daß durch Konfiguration und Zusammenbau vorgefertigter Komponenten spezialisierte ORBs hergestellt werden können. Unglücklicherweise erfordert aber die Implementierung eines CORBA–Systems selbst unter Verwendung eines mächtigen Frameworks einen erheblichen Aufwand, wie am Beispiel von TAO [55] deutlich wird.

Kapitel 4

Interoperabilität

Einer der größten Nachteile von CORBA vor der Version 2.0 war die fehlende Spezifikation von Mechanismen, die die Interoperabilität zwischen CORBA–Produkten verschiedener Hersteller sicherstellen. Mit CORBA 2.0 wurde eine allgemeine Interoperabilitäts–Architektur vorgestellt. Sie bietet sowohl Unterstützung für Interoperabilität zwischen verschiedenen CORBA–ORBs als auch zwischen CORBA–ORBs und anderen Middleware–Plattformen wie beispielsweise OSF–DCE und DCOM von Microsoft.

In diesem Kapitel wird zunächst auf das der Interoperabilität in CORBA zugrundeliegende Modell und auf die darauf basierenden, von CORBA spezifizierten Protokolle eingegangen. Danach wird ein Entwurf für die Unterstützung dieser Mechanismen in MICO auf Basis des in Kapitel 3 beschriebenen Mikrokern–ORB vorgestellt.

4.1 Modell

In Hinblick auf Interoperabilität werden Objekte als unteilbares Ganzes betrachtet. So sind beispielsweise alle Methoden eines Objektes über den gleichen Zugriffsmechanismus aufrufbar. Jedoch können sich Objekte untereinander bezüglich der Interoperabilitäts–Eigenschaften erheblich unterscheiden. Beispiele für solche Eigenschaften sind:

- Middleware–Plattform, in der das Objekt existiert
- CORBA–Implementierung, in der das Objekt existiert
- Protokoll, mit dem auf das Objekt zugegriffen werden kann
- Sicherheitsanforderungen, unter denen auf das Objekt zugegriffen werden kann

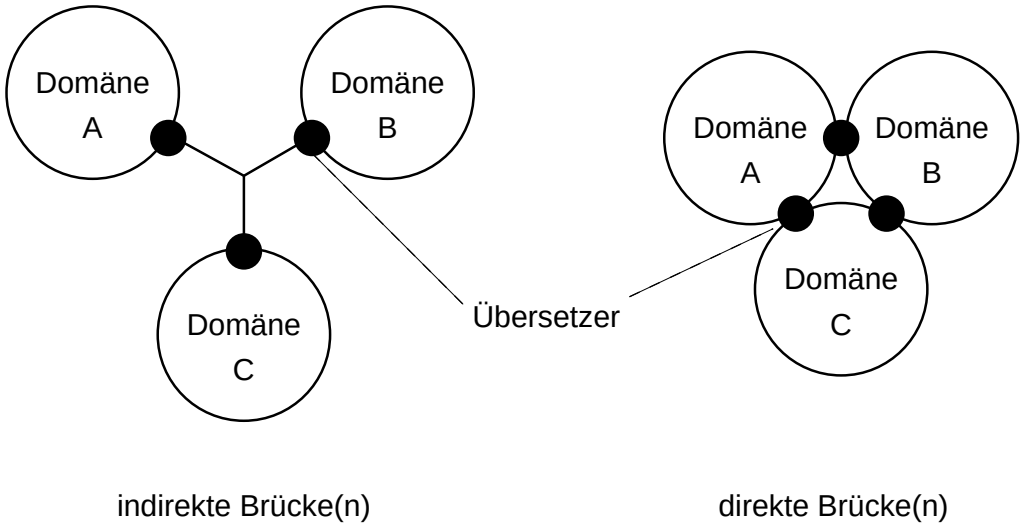


Abbildung 4.1: direkte vs. indirekte Brücke(n)

Daraus wird deutlich, daß Interoperabilität nicht nur eng mit den zur Kommunikation zwischen Objekten verwendeten Protokollen verbunden ist, sondern auch mit den zur Anwendung kommenden Object Services (wie beispielsweise Security– und Transaction–Service).

Zur Diskussion von Interoperabilität werden Objekte mit gleichen Werten bezüglich einer Interoperabilitäts–Eigenschaft zu Gruppen, sogenannten *Domänen*, zusammengefaßt. Objekte befinden sich entweder vollständig innerhalb oder vollständig außerhalb einer Domäne, was die oben erwähnte „Unteilbarkeitseigenschaft“ von Objekten widerspiegelt. Objekte können sich in keiner, einer oder mehr als einer Domäne befinden. Dadurch bedingt können Domänen disjunkt sein, sich überlappen oder einander enthalten.

Interoperabilität beschäftigt sich nun damit, wie Objekte aus verschiedenen Domänen miteinander kommunizieren können. Dazu ist es formal gesehen notwendig, eine bijektive Abbildung zwischen den verschiedenen Verhaltensweisen innerhalb der Domänen zu realisieren. Eine solche Abbildung bezeichnet man als *Brücke*. Die durch eine Brücke realisierte Abbildung muß bijektiv sein, da in einem objektorientierten System jedes aufgerufene Objekt potentiell auch ein aufrufendes Objekt ist, weil Objekreferenzen[1] als Parameter übergeben werden können.

Man unterscheidet zwei Arten von Brücken (siehe dazu auch Abbildung 4.1):

- indirekte Brücke
- direkte Brücke

[1] beispielsweise von Callback–Objekten

Bei einer *direkten Brücke* wird das Verhalten innerhalb einer Domäne direkt in das Verhalten innerhalb der anderen Domäne übersetzt. Dazu wird für jedes Paar von Domänen ein *Übersetzer* benötigt, der das Verhalten zwischen den beiden Domänen übersetzt. Die direkte Brücke weist folgende Eigenschaften auf:

+ ein Übersetzungsvorgang zwischen zwei Domänen

– Anzahl der Übersetzer quadratisch in der Anzahl der Domänen

Eine *indirekte Brücke* verwendet ein kanonisches Zwischenformat. Für jede Domäne existiert genau ein Übersetzer, der das Verhalten in der Domäne in das Zwischenformat und umgekehrt überführt. Dieser Ansatz hat folgende Eigenschaften:

+ Anzahl der Übersetzer linear in der Anzahl der Domänen

– zwei Übersetzungsvorgänge zwischen zwei Domänen

Die Übersetzer können auf verschiedenen *Ebenen* angeordnet sein. Man unterscheidet:

- Übersetzer auf ORB–Ebene
- Übersetzer auf Anwendungsebene

Während *Übersetzer auf ORB–Ebene* Bestandteil des CORBA–Systems sind und meist vom CORBA–Hersteller bereitgestellt werden, sind *Übersetzer auf Anwendungsebene* außerhalb des ORB angeordnet und können vom Benutzer unter Verwendung von DII, DSI und Interface Repository implementiert werden. Bedingt dadurch, daß alle Aufrufe DII und DSI durchlaufen, sind Übersetzer auf ORB–Ebene wesentlich effizienter als solche auf Anwendungsebene.

4.2 Inter–ORB–Protokolle

Aufbauend auf dem in Abschnitt 4.1 beschriebenen Interoperabilitäts–Modell definiert die CORBA–Spezifikation Datenformate und Protokolle zur Konstruktion von Brücken zwischen Objekt–Domänen. Dabei wird Interoperabilität auf zwei Ebenen unterstützt:

- zwischen verschiedenen CORBA–ORBs
- zwischen CORBA–ORBs und anderen Middleware–Plattformen

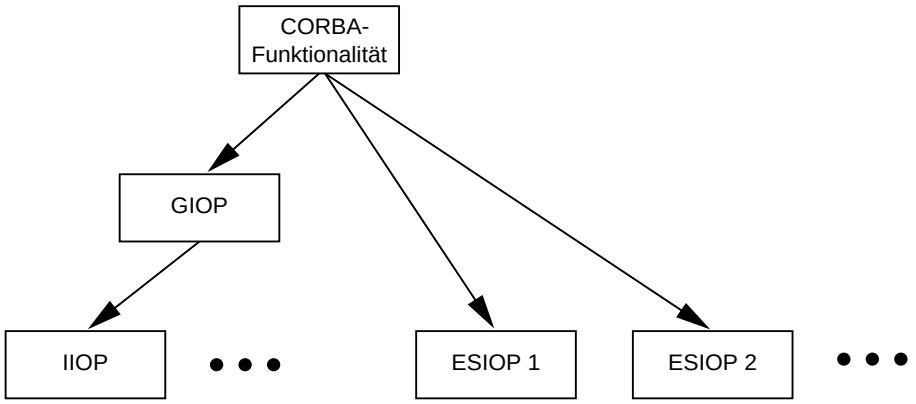


Abbildung 4.2: Interoperabilitäts–Unterstützung durch CORBA

Für die Interoperabilität zwischen verschiedenen ORBs sorgt das *General Inter–ORB Protocol (GIOP)*, während zur Kommunikation mit anderen Middleware–Plattformen sogenannte *Environment–Specific Inter–ORB Protocols (ESIOP)* zur Anwendungen kommen. GIOP ist an kein bestimmtes Transportprotokoll gebunden und kann daher nicht direkt implementiert werden; dafür müßen Abbildungen von GIOP auf Transportprotokolle definiert werden. Das *Internet Inter–ORB Protocol (IIOP)* beispielsweise definiert eine Abbildung von GIOP auf TCP/IP.

GIOP und ESIOPs bauen auf einer gemeinsamen Architektur zur Referenzierung von Objekten über Domänengrenzen hinweg auf, den sogenannten *Interoperable Object References (IORs)*. Die nächsten Abschnitte gehen näher auf IORs, GIOP und ESIOPs ein.

4.2.1 Interoperable Object References (IOR)

Eine wichtige Eigenschaft, in der sich Domänen unterscheiden können, ist die Repräsentation von Objektreferenzen. Um Interoperabiliät über Domänengrenzen hinweg zu gewährleisten, wird ein gemeinsames Zwischenformat für Objektreferenzen benötigt. Eigentlich ist die Spezifikation eines solchen Zwischenformates Sache eines Protokolls wie GIOP (wie die Spezifikation eines Formats zur Darstellung von IDL–Datentypen auch), jedoch gibt CORBA ein allgemeines Framework zur Darstellung von Objektreferenzen vor, das auf das jeweils verwendete Protokoll abgebildet werden muß. Der Aufbau der IORs orientiert sich an den von einer Brücke benötigten Informationen:

- Ist das Objekt NIL?
- Welchen Typ hat das Objekt?
- Mit welchen Protokollen kann auf das Objekt zugegriffen werden?
- Welche Object Services sind beteiligt?

Aufbauend auf diesem Informationsmodell werden IORs mittels CORBA–IDL spezifiziert. Durch Verwendung von IDL geschieht die Abbildung von IORs auf spezifische Protokolle wie GIOP automatisch durch die Regeln zur Abbildung von IDL–Datentypen auf eine Transfer–Syntax.

```
// IDL
module IOP {
    typedef unsigned long ProfileId;

    struct TaggedProfile {
        ProfileId tag;
        sequence<octet> profile_data;
    };

    struct IOR {
        string type_id;
        sequence<TaggedProfile> profiles;
    };

    ...
};
```

Da die Menge der verwendbaren Protokolle und Object Services potentiell erweiterbar ist, ist die Struktur von vielen in einer IOR enthaltenen Informationen, wie beispielsweise Adressen, beim Entwurf des IOR Frameworks unbekannt. Deshalb besteht eine IOR nur aus einem nicht näher strukturierten Typ (`type_id`) und einer Menge von Profilen (`profiles`). Jedes Profil enthält alle Informationen, die für den Zugriff auf das zugehörige Objekt mittels eines ganz speziellen Zugriffsmechanismus notwendig sind. Kann auf ein Objekt mittels mehrerer Mechanismen zugegriffen werden, so enthält die IOR für jeden Mechanismus ein Profil. IORs mit null Profilen repräsentieren ein NIL–Objekt. Ein Profil besteht aus einem global von der OMG vergebenen Identifizierer (`tag`) und unstrukturierten Daten (`profile_data`). Jedes Inter–ORB–Protokoll muß den Aufbau dieser Daten spezifizieren.

Um IORs einfach weitergeben zu können (beispielsweise mittels Email), beschreibt der Standard einen Mechanismus zur Umwandlung von IORs in Zeichenketten und umgekehrt.

4.2.2 General Inter–ORB Protocol (GIOP)

Interoperabilität zwischen verschiedenen CORBA–ORBs basiert auf dem GIOP, das jeder standardkonforme ORB unterstützen muß. GIOP ist damit das Zwischenformat zur Konstruktion indirekter Brücken zwischen allen standardkonformen ORBs. Aktuelle CORBA–Produkte verwenden GIOP häufig bereits als

ihr natives Kommunikationsprotokoll, so daß für die Zusammenarbeit von ORBs verschiedener Hersteller häufig gar keine Brücke mehr notwendig ist.

GIOP wurde für die Verwendung über beliebige verbindungsorientierte Protokolle (wie beispielsweise TCP) entworfen und besteht aus zwei Komponenten:

- *Common Data Representation (CDR)* definiert eine Abbildung der IDL–Datentypen auf einen Byte–Strom.
- *GIOP–Nachrichtenformate* bilden die ORB–Funktionalität (wie beispielsweise Methodenaufrufe) auf Protokollnachrichten ab.

Die Protokollnachrichten sind selbst durch CORBA–IDL spezifiziert und werden ebenfalls mittels CDR auf einen zu übertragenden Byte–Strom abgebildet. GIOP ist abstrakt, da es nicht an ein bestimmtes Transportprotokoll gebunden ist. Es werden in der Spezifikation lediglich einige Annahmen über die Eigenschaften verwendbarer Transportprotokolle gemacht:

- *verbindungsorientiert:* die Verbindung muß vor der Übertragung von Nachrichten geöffnet und danach geschlossen werden
- *zuverlässig:* Nachrichten werden nicht vertauscht und genau einmal zugestellt
- *Byte–Strom–orientiert:* keine Größenbeschränkungen für zu übertragende Nachrichten und keine Paketgrenzen
- *Fehler–Notifikation:* tritt eine Störung auf, so wird der Benutzer darüber informiert

Um GIOP über ein Transportprotokoll zu verwenden, das diese Eigenschaften erfüllt, muß eine Abbildung von GIOP auf dieses Protokoll definiert werden. Das *Internet Inter–ORB Protocol (IIOP)* ist eine Abbildung von GIOP auf TCP/IP, das jeder standardkonforme ORB bereitstellen muß. Neben IIOP werden vom CORBA–Standard derzeit keine weiteren GIOP–Mappings definiert, jedoch verwenden einige Produkte spezielle Mappings. Beispielsweise ist das in TAO [55] verwendete *Realtime Inter–ORB Protocol (RIOP)* eine Abbildung von GIOP auf ATM. Neben der Verwendung des Transportprotokolls definiert jedes GIOP–Mapping den Aufbau der in IORs verwendeten Protokollprofile (siehe Abschnitt 4.2.1). Beispielsweise enthält ein IIOP–Protokollprofil Internet–Adresse und TCP–Portnummer des Kommunikationsendpunktes, unter denen das zur IOR gehörende Objekt zu finden ist.

4.2.3 Environment–Specific Inter–ORB Protocols

Im Gegensatz zum GIOP dienen die ESIOPs nicht zur Kommunikation zwischen ORBs verschiedener Hersteller, sondern zur Interoperabilität zwischen CORBA–ORBs und anderen Middleware–Plattformen wie beispielsweise DCE oder DCOM. ESIOPs erlauben damit die Integration bereits existierender Verteilungsplattformen in CORBA. Die Bereitstellung von ESIOPs durch CORBA–Hersteller ist optional.

Jedes ESIOP definiert eine Abbildung von IDL–Datentypen auf Datentypen der anzubindenden Middleware Plattform. Ebenso wird eine Abbildung der ORB–Funktionalität auf Kommunikationsmechanismen der anderen Plattform definiert. Schließlich legt jedes ESIOP das Format der IOR–Protokollprofile fest.

Das erste von der OMG spezifizierte ESIOP war das mit CORBA 2.0 eingeführte *DCE Common Inter–ORB Protocol (DCE–CIOP)*, das die einfache Integration von CORBA und OSF–DCE–Applikationen erlaubt.

4.3 Design der MICO–Interoperabilität

Gemäß der CORBA–Spezifikation muß jedes standardkonforme CORBA–Produkt IIOP zur Verfügung stellen. Als erweiterbare CORBA–Plattform stellt MICO ein allgemeines Framework zur Realisierung von Interoperabilitäts–Mechanismen zur Verfügung. Aufbauend darauf wird ein Entwurf zur Unterstützung von GIOP in MICO vorgestellt. Die Verwendung des Frameworks ermöglicht es, nicht nur ein bestimmtes GIOP–Mapping wie beispielsweise IIOP zu implementieren, sondern GIOP als allgemeinen Mechanismus bereitzustellen.

4.3.1 Framework

Das Interoperabilitäts–Framework stellt neben einigen allgemeinen Hilfsmitteln Mechanismen auf der OSI Transport–, Darstellungs– und Anwendungsschicht bereit. Die meisten dieser Mechanismen sind abstrakt in dem Sinne, daß sie an kein bestimmtes Protokoll oder Datenformat gebunden sind. Erst durch Vererbung und Implementierung zusätzlicher Methoden werden die Mechanismen an ein konkretes Protokoll oder Datenformat gebunden. Abbildung 4.3 zeigt ein UML–Diagramm der abstrakten Komponenten des Frameworks.

Speicherverwaltung

Grundlage für ein performantes CORBA–System ist eine effiziente Speicherverwaltung. Insbesondere ist es wichtig, das Kopieren von Daten soweit wie möglich zu vermeiden. Einige wichtige Stellen, an denen potentiell Daten kopiert werden müssen, sind:

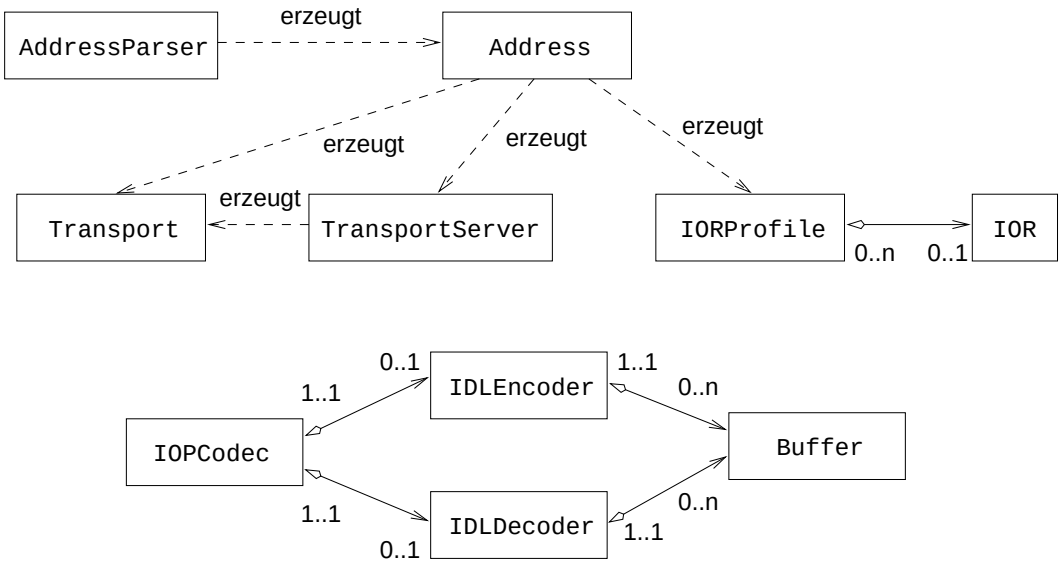


Abbildung 4.3: UML–Diagramm des Interoperabilitäts–Frameworks

- zwischen Netzwerkhardware und Interoperabilitätsmodulen
- zwischen Interoperabilitätsmodulen und ORB–Kern
- zwischen ORB–Kern und Objekt–Adaptern
- zwischen Objekt–Adaptern und Objekt–Implementierungen
- beim Marshalling von Daten

Beim Marshalling von Daten ist oftmals zu Beginn nicht bekannt, wieviel Speicherplatz benötigt wird, so daß gegebenenfalls während des Marshallings bereits gemarshallte Daten in einen größeren Speicherbereich umkopiert werden müssen.

Um das Kopieren von Daten so weit wie möglich zu beschränken, stellt das Framework sogenannte `Buffer`–Objekte bereit, die einen zusammenhängenden Speicherbereich verwalten. Dieser Speicherbereich kann durch enge Zusammenarbeit mit der dynamischen Speicherverwaltung ohne zu Kopieren vergrößert werden. Außerdem verfügt jeder `Buffer` über einen *Referenzzähler*. Damit kann überall dort, wo Speicher potentiell kopiert werden müßte, einfach der Referenzzähler erhöht und ein Verweis auf den `Buffer` übergeben werden. Wird der `Buffer` nicht mehr benötigt, so wird der Referenzzähler dekrementiert. Erreicht der Referenzzähler dabei Null, wird der `Buffer` gelöscht.

Transportschicht

Auf der Transportschicht stellt das Framework geeignete Abstraktionen für verbindungsorientierte Transportprotokolle bereit. `Transport`– und

`TransportServer`–Objekte modellieren die Semantik eines verbindungsorientierten Protokolls, wie es von GIOP erwartet wird. Sie sind ebenso wie GIOP abstrakt in dem Sinne, daß sie an kein bestimmtes Protokoll gebunden sind. Durch Vererbung können diese abstrakten Objekte an ein bestimmtes Protkoll, wie beispielsweise TCP gebunden werden.

`Transport`–Objekte modellieren einen Kommunikationsendpunkt mit folgenden Operationen:

- Herstellen/Auflösen einer Verbindung zu einem entfernten `TransportServer`
- Versenden von Daten aus einem `Buffer`
- Empfangen von Daten in einen `Buffer`

Zum Senden und Empfangen von Daten arbeiten `Transport`–Objekte unsichtbar für den Anwender eng mit dem in Abschnitt 3.3.4 beschriebenen Scheduler zusammen. `TransportServer`–Objekte modellieren spezielle Kommunikationsendpunkte, die serverseitig zum Aufbau neuer Verbindungen dienen. Stellt ein entferntes `Transport`–Objekt eine Verbindung zu einem `TransportServer`–Objekt her, so erzeugt das `TransportServer`–Objekt ein neues `Transport`–Objekt, das zum Austausch von Daten mit dem entfernten `Transport` Objekt verwendet werden kann.

`Address`–Objekte dienen zur Adressierung von Kommunikationsendpunkten. Sie bieten folgende Funktionen:

- Umwandlung einer Adresse in eine Zeichenkette
- Umwandlung einer Zeichenkette in eine Adresse
- Factory für `Transport`, `TransportServer` und `IORProfile`–Objekte

`AddressParser` sind Hilfsobjekte zur Rückumwandlung von Zeichenketten in `Address`–Objekte. Für jeden Adreßtyp muß eine von `AddressParser` abgeleitete Klasse bereitgestellt werden.

`Address`–Objekte dienen als Factory [14] zur Erzeugung von `Transport`–, `TransportServer`– und `IORProfile`–[2] Objekten.

Liegt beispielsweise eine Adresse in Form einer vom Benutzer eingegebenen Zeichenkette vor, so kann diese unter Verwendung eines `AddressParser` in ein `Address`–Objekt umgewandelt werden. Ohne zu wissen, zu welchem speziellen Protokoll die Adresse gehört, kann das `Address`–Objekt ein `Transport`–Objekt

[2] `IORProfile`–Objekte modellieren die in Abschnitt 4.2.1 beschriebenen Protokollprofile der interoperablen Objektreferenzen und werden vom Framework als Mechanismus der Anwendungsschicht bereitgestellt.

erzeugen, daß das zur Adresse passende Protokoll implementiert. Schließlich kann man eine Verbindung zum Kommunikationsendpunkt mit dieser Adresse herstellen lassen und Daten austauschen, ohne das verwendete Protokoll zu kennen.

Auch `Address`– und `AddressParser`–Objekte sind abstrakt und müssen erst durch Vererbung an ein bestimmtes Protokoll gebunden werden.

Darstellungsschicht

Auf der Darstellungsschicht bietet das Framework Unterstützung für Marshalling und Demarshalling von IDL–Datentypen. `IDLEncoder`–Objekte wandeln IDL–Datentypen in einen Byte–Strom um, der in einem `Buffer`–Objekt abgelegt wird. `IDLDecoder`–Objekte lesen einen Byte–Strom aus einem `Buffer`–Objekt und wandeln ihn in IDL–Datentypen um. Auch `IDLEncoder`– und `IDLDecoder`–Objekte sind abstrakt und müssen durch Vererbung an ein bestimmtes Marshalling–Format wie beispielsweise CDR gebunden werden.

Anwendungsschicht

Die Anwendungsschicht des Frameworks bietet Unterstützung für das Erzeugen und Dekodieren von Nachrichten eines Inter–ORB–Protokolls, wie beispielsweise GIOP. `IOPCodec`–Objekte verfügen dazu unter anderem über Operationen zum Erzeugen und Dekodieren folgender Nachrichten:

1. Invocation–Request
2. Invocation–Response
3. Cancel–Request

Diese dienen zur Durchführung von Methodenaufrufen (1, 2) und zum Abbruch eines in Bearbeitung befindlichen Methodenaufrufes (3). Zur Erzeugung von Nachrichten wird ein `IDLEncoder`–Objekt verwendet, das die Nachricht in einem `Buffer`–Objekt als Byte–Strom ablegt. Beim Empfang einer Nachricht wird der aus einem `Buffer`–Objekt gelesene Byte–Strom mittels eines `IDLDecoder`–Objektes dekodiert. `IOPCodec`–Objekte sind abstrakt und müssen durch Vererbung an ein bestimmtes Nachrichtenformat gebunden werden.

Ebenfalls Bestandteil der Anwendungsschicht sind `IOR`–Objekte, die die in Abschnitt 4.2.1 beschriebenen interoperablen Objektreferenzen modellieren. Jedes `IOR`–Objekt enthält neben Typ und Identität eines CORBA–Objektes eine Menge von Protokollprofilen. Letztere werden im Framework durch `IORProfile`–Objekte modelliert und müssen durch Vererbung an ein bestimmtes Protokoll gebunden werden. Da Protokollprofile minimal die Adresse enthalten, unter der das zugehörige CORBA–Objekt gefunden werden kann, können `Address`–Objekte als Factory zur Erzeugung passender `IORProfile`–Objekte verwendet werden.

4.3.2 GIOP

Das im vorangegangenen Abschnitt beschriebene abstrakte Framework ermöglicht es, GIOP ebenfalls als abstrakten Mechanismus unabhängig von einem bestimmten Transportprotokoll wie beispielsweise TCP zu implementieren. Um ein bestimmtes GIOP–Mapping wie beispielsweise IIOP bereitzustellen, müssen dann nur die entsprechenden abstrakten Komponenten des Frameworks durch Vererbung an konkrete Protokolle gebunden werden. Für IIOP werden beispielsweise folgende Komponenten benötigt:

Komponente	erbt von	Beschreibung
`TCPTransport`	`Transport`	TCP–Kommunikationsendpunkt
`TCPTransportServer`	`TransportServer`	TCP–Kommunikationsendpunkt zum Aufbau neuer Verbindungen
`InetAddress`	`Address`	Internet Adresse, enthält IP–Adresse und Portnummer
`InetAddressParser`	`AddressParser`	Parser für Internet–Adressen
`CDREncoder`	`IDLEncoder`	Komponente für Marshalling von IDL–Datentypen gemäß CDR
`CDRDecoder`	`IDLDecoder`	Komponente für Demarshalling von IDL–Datentypen gemäß CDR
`IIOPProfile`	`IORProfile`	IIOP–Protokollprofil, enthält IP–Adresse, Portnummer, IIOP–Version und Objekt–ID
`GIOPCodec`	`IOPCodec`	Komponente zur Erzeugung und Dekodierung von GIOP–Nachrichten

Abbildung 4.4 zeigt, wie die Unterstützung für GIOP in MICO mittels der zwei Komponenten *GIOP–Klient* und *GIOP–Server* realisiert wird. Der GIOP–Klient ist aus Sicht des ORB eine aufrufausführende Komponente, die wie ein Objekt–Adapter Methodenaufrufe vom ORB entgegenimmt, sie aber dann in GIOP–Nachrichten umwandelt und an einen GIOP–Server schickt. Der GIOP–Server ist eine aufrufgenerierende Komponente, die Nachrichten empfängt, in Methodenaufrufe umwandelt und an den lokalen ORB übergibt. Die nächsten beiden Abschnitte befassen sich näher mit dem Aufbau von GIOP–Klient und –Server.

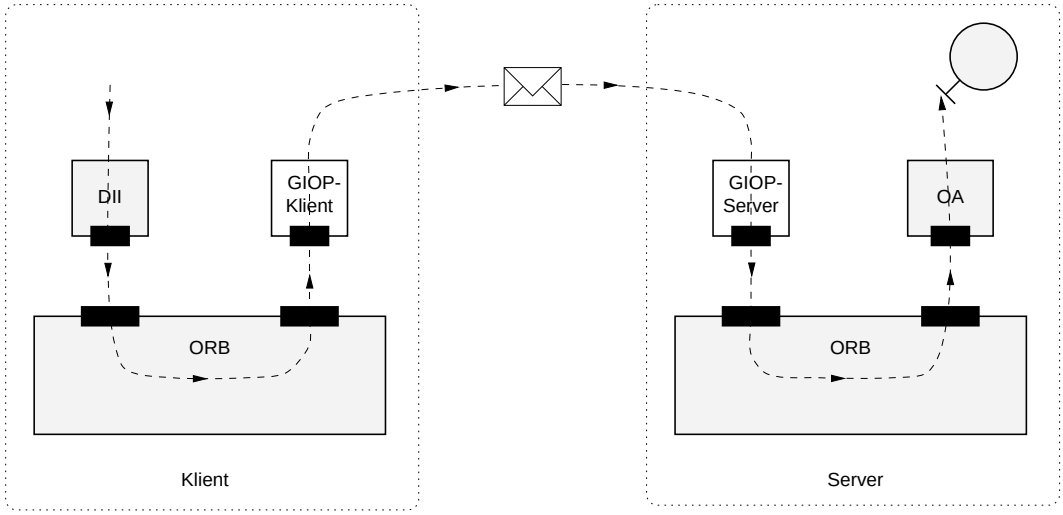


Abbildung 4.4: GIOP–Unterstützung in MICO

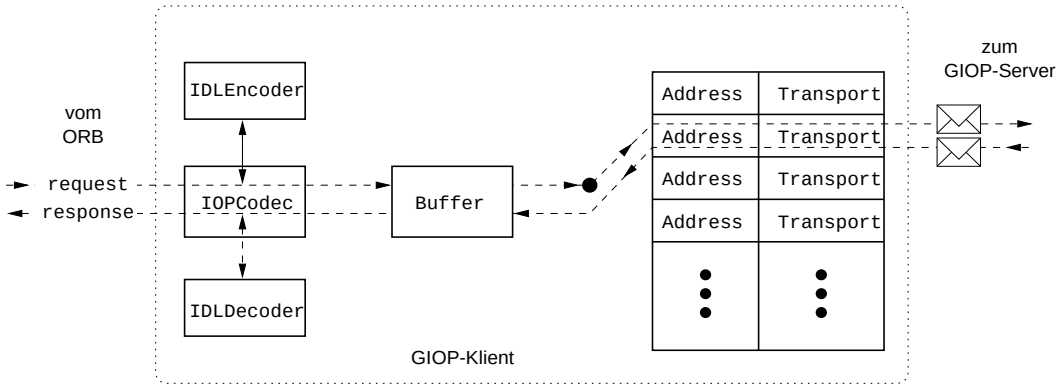


Abbildung 4.5: GIOP–Klient

GIOP–Klient

Abbildung 4.5 zeigt, wie der GIOP–Klient unter Verwendung von Komponenten des Frameworks realisiert wird. Ein vom ORB eintreffender Methodenaufruf (`request` in Abbildung 4.5) wird mit Hilfe eines `IOPCodec` in eine Protokollnachricht umgewandelt und in einem `Buffer` als Byte–Strom abgelegt. Für das Marshalling der Datentypen verwendet der `IOPCodec` einen geeigneten `IDLEncoder`. Verbindungen zu GIOP–Servern werden im Klienten durch `Transport`–Objekte repräsentiert. Da aus Effizienzgründen die Verbindung zu einem GIOP–Server über die Dauer eines Methodenaufrufes hinaus bestehen bleibt, so daß spätere Methodenaufrufe zum gleichen GIOP–Server die bereits bestehende Verbindung nutzen können, muß der GIOP–Klient unter Verwendung einer Tabelle Buch über die bereits bestehenden Verbindungen und die Adressen der zugehörigen Server führen. Um eine Nachricht abzuschicken, extrahiert der GIOP–Klient aus der Ob-

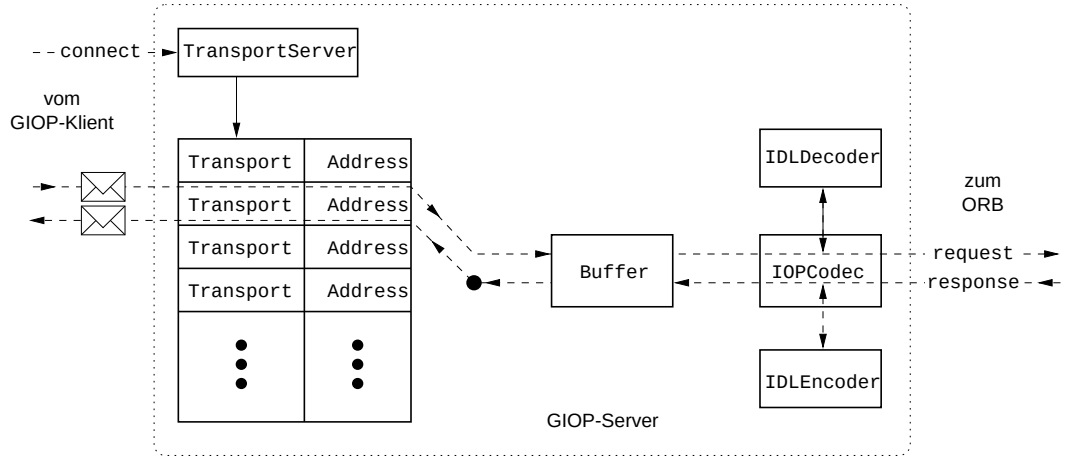


Abbildung 4.6: GIOP–Server

jektreferenz des Zielobjektes des Methodenaufrufes die Adresse des zugehörigen GIOP–Servers und konsultiert seine Tabelle. Ist die Adresse nicht in der Tabelle verzeichnet, wird eine neue Verbindung geöffnet und in der Tabelle eingetragen. Schließlich wird die Nachricht über das `Transport`–Objekt abgeschickt.

Die Ergebnisse des Methodenaufrufes werden in einem `Buffer` abgelegt und vom `IOPCodec` unter Zuhilfenahme eines `IDLDecoder` dekodiert. Schließlich wird das Ergebnis (`response` in Abbildung 4.5) dem ORB bekanntgegeben.

GIOP–Server

Der GIOP–Server ist das Gegenstück zum GIOP–Klient. Auch er muß unter Verwendung einer Tabelle Buch über zu GIOP–Klienten bestehende Verbindungen führen. Will ein GIOP–Klient eine Nachricht an einen GIOP–Server schicken, muß dieser zunächst eine Verbindung zum Server herstellen. Dazu schickt er wie in Abbildung 4.6 gezeigt eine Verbindungsanfrage an den Server, die beim `TransportServer` Objekt im GIOP–Server eintrifft (`connect` in Abbildung 4.6). Daraufhin wird im Server für die neue Verbindung ein `Transport`–Objekt erzeugt und in die Verbindungstabelle eingetragen. Nun ist der GIOP–Klient in der Lage, mit dem GIOP–Server Nachrichten auszutauschen.

Trifft beim Server eine Nachricht ein, wird sie vom `IOPCodec` unter Zuhilfenahme eines `IDLDecoder` dekodiert und der resultierende Methodenaufruf (`request` in Abbildung 4.6) dem ORB zur Ausführung übergeben. Teilt der ORB später dem GIOP–Server das Ergebnis des Methodenaufrufes (`response` in Abbildung 4.6) mit, wird dieses vom `IOPCodec` unter Verwendung eines `IDLEncoders` in einen `Buffer` gemarshallt. Schließlich konsultiert der Server seine Verbindungstabelle, um das `Transport`–Objekt zu finden, über das die Nachricht an den GIOP–Klient abgeschickt werden muß. Im Unterschied zum Klient kann der GIOP–Server davon ausgehen, daß ein entsprechender Eintrag bereits in der Tabelle vorhanden

ist, da die Verbindung schon zum Empfang der Methodenaufruf–Nachricht vom Klienten geöffnet wurde.

4.4 Zusammenfassung, Bewertung und Alternativen

Die Konstruktion von Interoperabilitäts–Komponenten wird von MICO durch ein abstraktes Framework unterstützt, das nicht an bestimmte Protokolle gebunden ist. Auf Basis dieses Frameworks kann GIOP in MICO ohne Bindung an bestimmte Transportprotokolle wie beispielsweise TCP realisiert werden. Zur Unterstützung eines bestimmten GIOP–Mappings wie IIOP müssen lediglich die abstrakten Komponenten des Frameworks durch Vererbung an konkrete Protokolle gebunden werden, was auch nachträglich durch den Anwender geschehen kann.

Die GIOP–Unterstützung wird durch zwei Services außerhalb des in Kapitel 3 vorgestellten Mikrokern–ORB realisiert: der *GIOP–Klient* transformiert vom ORB entgegengenommene Methodenaufrufe in Netzwerkpakete, die der *GIOP–Server* empfängt und in Methodenaufrufe auf den ORB umwandelt. Dieser Ansatz weist folgende Eigenschaften auf:

+ einfache Integration neuer GIOP–Mappings
+ einfache Integration neuer Inter–ORB–Protokolle (z.B. ESIOPs)
– erhöhter Verwaltungsaufwand

Mit erhöhtem Verwaltungsaufwand ist gemeint, daß für jeden Methodenaufruf ein Lookup in der Verbindungstabelle notwendig ist. In einer konventionellen ORB–Implementierung, bei der die Interoperabilitätsmodule Bestandteil des ORB sind, ließe sich die Zuordnung zu den `Transport`–Objekten direkt in den Stub–Objekten abspeichern, so daß keine Tabelle und damit auch kein Lookup notwendig ist. Durch Verwendung von geeigneten Datenstrukturen (beispielsweise Hash–Tabellen) ist dieser Nachteil in der Praxis aber vernachlässigbar.

Kapitel 5

Objekt–Adapter

Objekt–Adapter entkoppeln objektspezifisches Verhalten vom ORB–Kern. Es wäre auch möglich gewesen, CORBA ohne die Verwendung von Objekt–Adaptern zu realisieren, indem der ORB direkt die Funktionalität von Objekt–Adaptern bereitstellt. Das würde jedoch dazu führen, daß der ORB alle denkbaren Formen von Objekten unterstützen müßte und dadurch groß, langsam und kompliziert anzuwenden wäre. Alternativ könnte der ORB nur einen speziellen Typ von Objekten unterstützen, was jedoch seine Verwendung stark einschränken würde. Objekt–Adapter tragen also dazu bei, das CORBA–System zugleich klein und flexibel zu halten.

In diesem Kapitel wird zunächst ein Überblick über die Funktionalität von Objekt–Adaptern gegeben. Darauf folgend werden einige Beispiele für Objekt–Adapter vorgestellt, darunter auch der *Basic Object Adapter (BOA)*, der von jeder standardkonformen CORBA–Implementierung bereitgestellt werden muß. Schließlich wird aufbauend auf den in Kapitel 3 vorgestellten Mechanismen das Design des MICO–BOA vorgestellt.

5.1 Terminologie

Objekt–ID: ist Teil der Identität eines CORBA–Objektes und wird entweder vom Objekt–Adapter oder vom Anwender vergeben. Objekt–IDs sind in der Regel nur unter den CORBA–Objekten in einem Adreßraum eindeutig. Anhand der Objekt–ID kann der für das zugehörige Objekt zuständige Objekt–Adapter bestimmt werden, der wiederum anhand der Objekt–ID das Zielobjekt eines Methodenaufrufes auswählt.

Objektreferenz: repräsentiert global eindeutig die Identität eines CORBA–Objektes. Teil der Objektreferenz sind die Objekt–ID sowie Adreßinformationen, die den Adreßraum, in dem sich das Objekt befindet, identifizieren. Mittels der Objektreferenz kann auf ein CORBA–Objekt aus einem anderen Adreßraum heraus zugegriffen werden.

Servant: verkörpert Zustand und Verhalten eines CORBA–Objektes. In prozeduralen Sprachen wie C ist ein Servant eine Ansammlung von Funktionen (Verhalten) und Datenfeldern (Zustand), in objektorientierten Sprachen wie C++ sind Servants Objekte.

CORBA–Objekt: ist die Gemeinsamkeit aus Identität, Zustand und Verhalten. Mit der bisher eingeführten Terminologie wird ein CORBA–Objekt durch eine Objektreferenz zusammen mit einem Servant repräsentiert. Es ist aber durchaus möglich, daß zu einem bestimmten Zeitpunkt zu einem CORBA–Objekt kein Servant existiert. Das bedeutet aber nicht, daß das Objekt dann keinen Zustand und Verhalten mehr hat, da ein Servant nur eine konkrete Darstellung von Zustand und Verhalten ist. Beispielsweise wäre es denkbar, daß der Zustand eines CORBA–Objektes in einer Datenbank abgelegt ist und ein Servant bei seiner Erzeugung den Zustand aus der Datenbank liest und bei seiner Zerstörung wieder dort ablegt.

Skelett: ein programmiersprachliches Konstrukt, das es einem Objekt–Adapter erlaubt, Methoden auf einem CORBA–Objekt auszuführen. Skelette werden in der Regel vom IDL–Compiler automatisch generiert. Außerdem stellt das DSI ein spezielles Skelett zu Verfügung, mit dem auch Methodenaufrufe auf Objekten durchgeführt werden können, für die keine vom IDL–Compiler generierten Skelette vorliegen.

Verkörperung (incarnation): verbindet ein CORBA–Objekt mit einem Servant und gibt damit dem abstrakten Objekt einen realen Körper.

Vergeistigung (etherealization): löst die Verbindung zwischen CORBA–Objekt und zugehörigem Servant.

Aktivierung: bringt ein CORBA–Objekt in einen Zustand, in dem es Methodenaufrufe ausführen kann. Damit ein Methodenaufruf tatsächlich durchgeführt werden kann, muß das Objekt mit einem Servant verknüpft sein.

Deaktivierung: bringt ein CORBA–Objekt in einen Zustand, in dem es keine Methodenaufrufe ausführen kann. Methodenaufrufe, die für ein gerade deaktiviertes Objekt eintreffen, können entweder abgelehnt oder solange verzögert werden, bis das Objekt wieder aktiviert wird. Phasen der Inaktivität sind in der Regel von kurzer Dauer und dienen oftmals zur Synchronisation des Zustands eines Objektes (beispielsweise mit einer Datenbank).

Erzeugung: ruft ein CORBA–Objekt ins Leben.

Zerstörung: beendet die Existenz eines CORBA–Objektes.

Objekttabelle: Komponente eines Objekt–Adapters, die eine Abbildung zwischen CORBA–Objekten (in Form von Objekt–IDs) und Servants definiert.

5.2 Funktionalität

Objekt–Adapter wurden als Komponenten des CORBA–Systems eingeführt, um der Vielfalt verschiedener denkbarer Objekttypen gerecht zu werden. Sie realisieren jegliche objektspezifische Funktionalität und entkoppeln damit objektspezifisches Verhalten vom ORB–Kern. Obwohl das eine Vielfalt verschiedener Objekt–Adapter mit ebenso verschiedener Funktionalität erwarten läßt, kann man doch zwei Gruppen von Funktionalität identifizieren, die jeder Objekt–Adapter zur Verfügung stellt:

- Verwaltung von Objekten
- Durchführung von Methodenaufrufen

5.2.1 Verwaltung von Objekten

Die mit der Verwaltung von Objekten für einen Objekt–Adapter anfallenden Aufgaben lassen sich am besten anhand des Lebenszyklus eines CORBA–Objektes beschreiben. Wie in Abbildung 5.1 gezeigt beginnt die Existenz eines CORBA–Objektes mit seiner Erzeugung. Für den Objekt–Adapter ist das mit der Generierung einer neuen Objektreferenz verbunden. Während der Objekt–Adapter den Typ und die Identität (in Form einer neu erzeugten Objekt–ID) des Objektes zur Objektreferenz beisteuert, werden Adressierungsinformationen vom ORB beigesteuert (siehe dazu auch Abschnitt 3.3.5). Danach existiert das neu erzeugte CORBA–Objekt zunächst als reine Identität ohne konkreten Zustand und konkretes Verhalten und ist inaktiv. Bereits in diesem Stadium können Klienten in Besitz der neu generierten Objektreferenz gelangen und Methodenaufrufe darauf initiieren. Da das Objekt jedoch inaktiv ist, können diese Methodenaufrufe zu diesem Zeitpunkt nicht durchgeführt werden. Der Objekt–Adapter kann mit diesen Methodenaufrufen auf verschiedene Art und Weise verfahren, darunter:

- *Zurückweisung:* der Methodenaufruf wird mit einer Fehlermeldung abgebrochen
- *Verzögerung:* der Methodenaufruf wird solange verzögert, bis das Objekt entweder aktiv wird (Methodenaufruf kann dann durchgeführt werden) oder bis das Objekt zerstört wird (Methodenaufruf wird mit Fehlermeldung abgebrochen)

Durch Aktivierung geht das Objekt schließlich in den aktiven Zustand, in dem es prinzipiell bereit ist, Methodenaufrufe durchzuführen. Es ist nur prinzipiell dazu fähig, weil ein Methodenaufruf die Kopplung des Objektes an einen Servant und damit an ein konkretes Verhalten und Zustand erfordert. Genau das geschieht bei der sogenannten Verkörperung, die das Objekt in einen Zustand versetzt, in

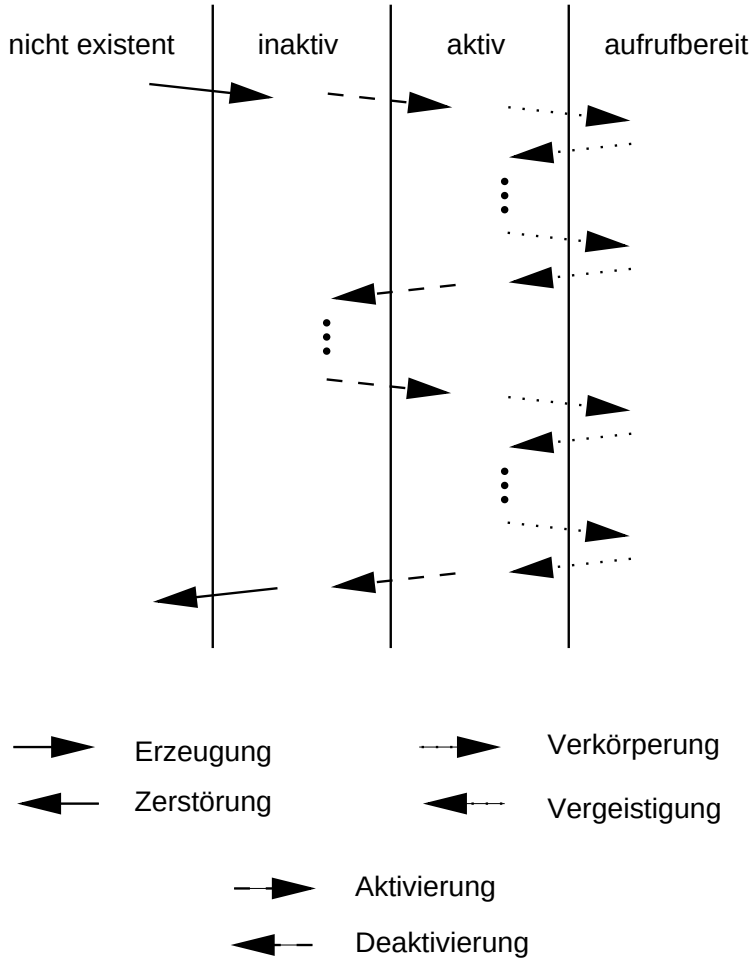


Abbildung 5.1: Lebenszyklus eines CORBA–Objektes

dem Methodenaufrufe tatsächlich durchgeführt werden können (*aufrufbereit* in Abbildung 5.1). Die Verkörperung kann unter anderem folgendermaßen ausgelöst werden:

- *bei Bedarf*, wenn ein Methodenaufruf für ein aktives Objekt eintrifft
- *vom Benutzer*

Die Verkörperung ist oftmals aus Sicht des Objekt–Adapters ein komplizierter Vorgang, da der Programmkode für den Servant meist in einem separaten Programm abgelegt ist. Falls dieses Programm nicht bereits aktiv ist, muß der Objekt–Adapter es starten. Außerdem sind viele verschiedene Möglichkeiten denkbar, wie der Servant auszuwählen ist. Der Objekt–Adapter muß dafür geeignete Schnittstellen zum Anwender bereitstellen. Folgende Zuordnungen von Servants zu Objekten sind denkbar:

- *1–zu–1:* ein Servant pro Objekt
- *1–zu–N:* mehrere Servants pro Objekt
- *N–zu–1:* ein Servant für mehrere Objekte

Durch die sogenannte Vergeistigung wird der Servant vom zugehörigen Objekt entkoppelt. Man unterscheidet zwei Typen von Objekten: Objekte während deren Lebenszeit genau eine Verkörperung und Vergeistigung stattfindet nennt man *transiente Objekte*. Falls mehr als eine Vergeistigung und Verkörperung während der Lebenszeit stattfindet, nennt man das Objekt *persistent*, weil bei einer Vergeistigung der Zustand des Objektes in geeigneter Form konserviert (*persistent* gemacht) und bei einer nachfolgenden Verkörperung wiederhergestellt werden muß. Das Konservieren des Zustands selbst ist Sache des Programmierers, die Unterstützung des Objekt–Adapters beschränkt sich in der Regel darauf, das Konservieren bzw. Wiederherstellen anzustoßen. CORBA bietet jedoch Unterstützung in Form des *Persistence Object Service (POS)* [35].

Persistente Objekte können während ihrer Lebenszeit an verschiedene Servants gekoppelt werden, die sich unter anderem in

- Verhalten
- Zustand
- Aufenthaltsort

unterscheiden können. Bei Veränderung von Verhalten oder Zustand eines Objektes während seiner Lebenszeit spricht man von *Evolution* [18], bei Änderung des Aufenthaltsortes von *Migration*.

Während seiner Lebenszeit kann ein Objekt mehrfach deaktiviert und wieder reaktiviert werden. In Abbildung 5.1 ist nicht erkennbar, daß ein Objekt auch ohne vorausgehende Vergeistigung deaktiviert werden kann. Die Existenz eines deaktivierten CORBA–Objektes endet mit dessen Zerstörung. Da in CORBA die Lebenszeit eines Objektes in keiner Weise mit der Lebenszeit von Stubs dieses Objektes verbunden ist, können Klienten auch noch nach dessen Zerstörung Methodenaufrufe auf einem Objekt initiieren, was jedoch mit einer Fehlermeldung quittiert wird.

Zusammenfassend stellt ein Objekt–Adapter im Zusammenhang mit der Verwaltung von Objekten folgende Funktionalität zur Verfügung:

- Generierung von Objektreferenzen
- Aktivierung/Deaktivierung von Objekten
  - Verzögern/Verwerfen von Methodenaufrufen

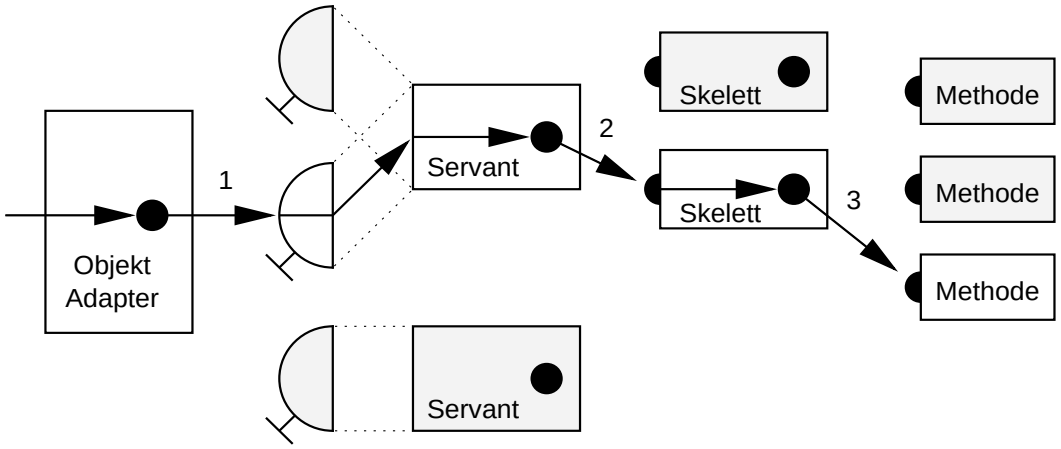


Abbildung 5.2: Dispatching eines Methodenaufrufes

- Verkörperung/Vergeistigung von Objekten
  - Starten von Programmen, die Servants implementieren
  - Zuordnung von Servants zu Objekten
  - Unterstützung für Persistenz von Objekten
  - Unterstützung für Evolution/Migration von Objekten

5.2.2 Durchführung von Methodenaufrufen

Auf einem Objekt im aufrufbereiten Zustand (siehe Abbildung 5.1) können Methodenaufrufe durchgeführt werden. Wie in Abbildung 5.2 gezeigt, durchläuft ein Methodenaufruf mehrere Dispatch–Vorgänge, bis er schließlich ausgeführt wird. Zunächst bestimmt der Objekt–Adapter anhand der Objektreferenz das Zielobjekt des Methodenaufrufes und den daran gekoppelten Servant (Schritt 1). Wie in der Abbildung gezeigt, ist es durchaus möglich, daß ein und derselbe Servant an verschiedene Objekte gekoppelt ist, die sogar unterschiedlichen Typs sein können. Da pro Objekttyp (IDL–Interface) vom IDL–Compiler je ein Skelett generiert wird, muß der Servant in diesem Fall unter diesen Skeletten das auswählen, das zum Typ des aufgerufenen Objektes gehört (Schritt 2). Das Skelett wählt schließlich die auszuführende Methode anhand des Methodennamens aus (Schritt 3), dekodiert die Eingabe–Parameter des Methodenaufrufes (siehe dazu auch Abschnitt 3.3.1) und führt schließlich den Methodenaufruf durch.

Skelette stellen die Verbindung zwischen Objekt–Adapter und vom Programmierer bereitgestellten Methodenimplementationen her. Ihre Struktur ist daher einerseits vom Objekt–Adapter und andererseits vom verwendeten Sprach–Mapping abhängig. Beispielsweise unterscheiden sich die Skelette in prozeduralen Programmiersprachen wie C erheblich von denen objektorientierter Programmiersprachen wie C++.

Eigenschaft	statisch	dynamisch
Performanz	hoch	gering
Handhabung	einfach	kompliziert
Skalierbarkeit	schlecht	gut
Typinfos zur Compilierzeit	benötigt	nicht benötigt

Tabelle 5.1: orthogonale Eigenschaften dynamischer und statischer Skelette

Man unterscheidet zwei Arten von Skeletten, *statische* und *dynamische*. Statische Skelette werden automatisch vom IDL–Compiler aus der in Form einer IDL vorliegenden Schnittstellenbeschreibung eines Objektes generiert, während dynamische Skelette vom Anwendungsprogrammierer implementiert werden müssen. Tabelle 5.1 zeigt die Eigenschaften beider Arten von Skeletten im Vergleich. Die Verwendung von statischen Skeletten ist einfach, weil sie vom IDL–Compiler automatisch generiert werden, während dynamische Skelette vollständig vom Anwendungsentwickler bereitgestellt werden müssen. Während die Anbindung dynamischer Skelette an Objekt–Adapter von der CORBA–Spezifikation mittels des *Dynamic Skeleton Interface (DSI)* vorgeschrieben wird, wird die Anbindung von statischen Skeletten an Objekt–Adapter dem CORBA–Implementierer überlassen. Dadurch sind zahlreiche Optimierungen möglich, die statische Skelette effizienter als dynamische machen. Die Unterschiede in der Skalierbarkeit rühren daher, daß für jeden Objekttyp ein separates statisches Skelett benötigt wird, während man mit einem dynamischen Skelett mehrere Objekttypen gleichzeitig handhaben kann. Prinzipiell ist es sogar möglich, ein dynamisches Skelett für alle Objekttypen zu verwenden. Da statische Skelette aus IDL–Beschreibungen generiert werden, müssen die Schnittstellen der Objekte natürlich zur Compilierzeit bekannt sein. Dynamische Skelette benötigen dagegen keine Typinformationen zur Compilierzeit.

Aufgrund ihrer Performanz und einfachen Handhabbarkeit werden für die meisten Anwendungen ausschließlich statische Skelette verwendet. Dynamische Skelette werden nur dann eingesetzt, wenn es auf gute Skalierbarkeit ankommt oder wenn zur Compilierzeit keine Typinformationen vorliegen.

Beispiel: Eine *Bridge* verbindet zwei verschiedene Middleware–Plattformen (CORBA und beispielsweise DCOM) miteinander in dem Sinne, daß die Ressourcen der einen Plattform der anderen verfügbar gemacht werden. Eine der Aufgaben einer Bridge ist die Weiterleitung von Methodenaufrufen von einer in die andere Plattform. Da die Bridge alle Typen von Objekten handhaben muß, können der Bridge zur Compilierzeit nicht die Schnittstellen aller Objekte bekannt sein. Die Bridge muß also DCOM–Objekte gegenüber CORBA durch Stellvertreterobjekte repräsentieren, die dynamische Skelette verwenden.

□

Einige Objekt–Adapter bieten darüber hinaus Sicherheitsmechanismen an, obwohl Sicherheit eigentlich nicht Gegenstand von Objekt–Adaptern sondern eines speziellen Object Service [35] ist. Beispielsweise ist mit dem Basic Object Adapter (BOA) eine Authentifizierung des Aufrufers möglich.

5.3 Beispiele für Objekt–Adapter

Der einzige momentan als Teil von CORBA spezifizierte Objekt–Adapter ist der *Basic Object Adapter (BOA)*. In einer der nächsten Versionen der CORBA–Spezifikation wird der BOA durch den *Portable Object Adapter (POA)* ersetzt. Sowohl BOA als auch POA sind Allzweck Objekt–Adapter in dem Sinne, daß sie für viele Standardanwendungen geeignet sind. Neben BOA und POA wird in den folgenden Abschnitten kurz auf zwei spezielle Objekt–Adapter–Konzepte, den *Library Object Adapter (LOA)* und den *Object–Oriented Database Object Adapter (OODB OA)* eingegangen.

5.3.1 Basic Object Adapter

Beim Entwurf des BOA wurde davon ausgegangen, daß Servants in Form von separaten Programmen vorliegen, sogenannten *Servern*. Dabei ist es durchaus möglich, daß ein Server mehrere Servants implementiert. Der BOA verfügt über eine Datenbank (genannt *Implementation Repository*), in der für jeden Server unter anderem verzeichnet ist, wie er gestartet werden kann. Das Starten eines Servers wird als *Serveraktivierung* bezeichnet. Genauer betrachtet ist mit einer Serveraktivierung die Erzeugung eines neuen Prozesses verbunden, in dem das Serverprogramm ausgeführt wird.

Serveraktivierung

Empfängt der BOA einen Methodenaufruf für ein Objekt, das noch nicht mit einem Servant verknüpft ist, so konsultiert er das Implementation Repository, um einen passenden Server zu ermitteln und zu starten. Eine Verkörperung ist also für den BOA unter Umständen mit einer Serveraktivierung verbunden. Um den verschiedenen Anforderungen an einen Allzweck–Objekt–Adapter gerecht zu werden, unterstützt der BOA vier verschiedene *Aktivierungsmodi*, die die Serveraktivierung beeinflussen:

Shared Server: In einem Serverprozeß können gleichzeitig mehrere CORBA–Objekte (auch verschiedenen Typs) existieren. Das ist der am häufigsten verwendete Aktivierungsmodus.

Persistent Server: Eine Spezialform eines *Shared Server*, bei der der Server nicht vom BOA, sondern vom Benutzer gestartet wird (beispielsweise durch

ein Shellscript beim Start eines Rechners). Dieser Aktivierungsmodus wird häufig dann verwendet, wenn das Starten des Servers sehr lange dauert, weil umfangreiche Initialisierungen nötig sind. Der Begriff „persistent“ hat nichts mit der in Abschnitt 5.2.1 beschriebenen Persistenz von Objekten zu tun.

Unshared Server: In einem Serverprozeß dieses Typs kann höchstens ein CORBA–Objekt existieren. Wird ein neues CORBA–Objekt erzeugt, so startet der BOA automatisch einen neuen Server.

Per–Method Server: Bei diesem Aktivierungsmodus startet der BOA für jeden Methodenaufruf einen neuen Server, der nur für die Dauer des Methodenaufrufes aktiv ist.

Sinn und Zweck der verschiedenen Aktivierungsmodi ist der verschiedene Grad von Parallelität bei der Methodenausführung. Unter der Annahme, daß in jedem Serverprozeß nur ein Kontrollfluß existiert (d.h., die Server sind single–threaded) können in einem *Shared Server* Methodenaufrufe nur squentiell ausgeführt werden, bei Verwendung des Aktivierungsmodus *Unshared Server* können Methodenaufrufe auf verschiedenen Objekten parallel ausgeführt werden und bei Verwendung von *Per–Method Servern* können sogar Methodenaufrufe auf einem Objekt parallel in verschiedenen Prozessen ausgeführt werden. Der BOA bietet jedoch keinerlei Möglichkeiten zur Synchronisation des Objektzustandes bei paralleler Ausführung von Methodenaufrufen; dafür muß gegebenenfalls der Programmierer sorgen.

Bewertung

Leider ist die Spezifikation des BOA sehr lückenhaft. Die Probleme damit lassen sich in drei Kategorien enteilen:

fehlende Funktionalität:

- *Multi–threading:* Oftmals ist es wünschenswert, auch innerhalb eines Serverprozesses Methoden durch Verwendung mehrere Threads parallel auszuführen. Damit in engem Zusammenhang müssen geeignete Mechanismen zur Synchronisation der Threads untereinander und mit dem BOA spezifiziert werden.
- *Objektgruppenverwaltung:* Zusammenfassung von eng verwandten Objekten in Gruppen, um beispielsweise Gruppen als Ganzes aktivieren zu können.

nicht spezifizierte Schnittstellen:

- *Implementation Repository:* Der Standard beschreibt eine Schnittstelle `ImplementationDef`, die einen Eintrag im Implementation Repository darstellt. Die Methoden dieser Schnittstelle sind jedoch nicht spezifiziert. Gar nicht erwähnt wird das Repository selbst, das die `ImplementationDef` Einträge erzeugt und verwaltet.
- *Skelette:* Der Aufbau der Skelette ist nicht klar definiert. In C++ beispielsweise sind Skelette Klassen, von denen der Anwender eine Implementierung ableitet. Unter anderem werden keine Namenskonventionen für diese Klassen spezifiziert.
- *Registrierung von Skeletten bei Servants*
- *Registrierung von Servants beim Objekt–Adapter*
- *Hooks für Objektpersistenz:* Schnittstellen, über die der BOA das Konservieren bzw. Wiederherstellen des Zustands eines persistenten Objektes auslöst.

unklare Semantik von spezifizierten Schnittstellen:

- *Synchronisation zwischen BOA und Objekt–Implementierung:* Bedeutung und Verhalten der Operationen `obj_is_ready()` und `impl_is_ready()` ist sehr vage spezifiziert.

Jeder BOA–Implementierer hat dadurch eine Reihe von Interpretationen und Ergänzungen zur Spezifikation vorzunehmen (siehe Abschnitt 5.4). Nahezu alle verfügbaren BOA–Implementierungen sind außerdem unvollständig, oftmals wegen nicht unterstützter Aktivierungsmodi. Dadurch bedingt ist serverseitiger Kode einer CORBA–Applikation unportabel zwischen verschiedenen CORBA–Produkten. Die OMG hat dieses Problem erkannt und in [36] zur Einreichung von Vorschlägen zur Behebung dieser Probleme entweder durch Verbesserung des BOA oder seine vollständige Ersetzung aufgerufen. Als Ergebnis dieses Aufrufes wurde die POA–Spezifikation [40] bei der OMG eingereicht, die den BOA in einer der nächsten Versionen der CORBA–Spezifikation ersetzen wird. Die aktuelle Version 2.1 der CORBA–Spezifikation fordert jedoch noch die Bereitstellung des BOA von einer standardkonformen CORBA–Implementierung.

5.3.2 Portable Object Adapter

Die POA–Spezifikation ist das Ergebnis langjähriger Erfahrungen diverser CORBA–Hersteller mit den Unzulänglichkeiten des BOA und proprietären Objekt–Adaptern. Aufgrund dieser Erfahrungen wurde die von einem Allzweck–Objekt–Adapter erwartete Funktionalität identifiziert. Die umfassende Definition

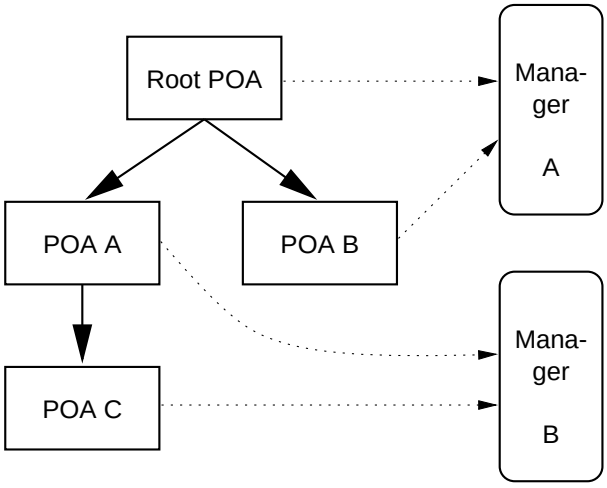


Abbildung 5.3: Portable Object Adapter

aller vom Anwender sichtbaren Schnittstellen und deren Semantik soll es ermöglichen, serverseitigen Kode von CORBA–Applikationen unabhängig vom verwendeten CORBA–Produkt zu machen.

Generelles Ziel beim Entwurf des POA war es, das Verhalten des POA soweit wie möglich durch den Anwender konfigurierbar zu machen, dabei jedoch geeignete Default–Einstellungen für Standardanwendungen zu bieten, um die Verwendung des POA nicht unnötig zu verkomplizieren. Hier nun ein kurzer Überblick über den POA.

Eine zentrale Eigenschaft des POA ist, daß in einem Serverprozeß mehrere POA–Instanzen gleichzeitig existieren können, um Objekte mit verschiedenen Eigenschaften innerhalb eines Servers zu unterstützen. Zur Konfiguration eines POA dienen folgende Komponenten:

- Policies
- Manager

Eine *Policy* steuert das Verhalten eines POA bezüglich eines ganz bestimmten Details. Beispielsweise regelt die sogenannte *Object ID Assignment Policy*, ob Objekt–IDs vom POA oder vom Benutzer vergeben werden. Im POA werden Policies durch spezielle *Policy Objekte* repräsentiert, die über ein einziges Attribut verfügen: eine Aufzählung aller möglicher Werte der Policy. *Manager* sind Objekte mit einer im Vergleich zu Policies komplexeren Funktionalität, die das Verhalten eines POA beeinflussen.

Um eine leichte Handhabung zu gewährleisten, sind, wie in Abbildung 5.3 dargestellt, alle POAs eines Serverprozesses in einer Hierarchie angeordnet. Zu Beginn existiert nur der sogenannte *Root POA*, der an der Spitze der Hierarchie steht. Er verfügt über Default–Policies und –Manager, so daß er ohne weitere

Konfiguration für Standard–Anwendungen benutzt werden kann. Der Benutzer kann nun weitere POAs als Kinder von bereits existierenden POAs erzeugen. Dabei können neue Manager und Policies angegeben werden, für nicht spezifizierte Policies und Manager werden die des Vater-POAs verwendet. Beispielsweise wurde bei der Erzeugung von *POA A* in Abbildung 5.3 angegeben, daß *Manager B* zu verwenden ist. Alle anderen POAs in der Abbildung erben die Manager von ihren Vätern. Diese „Vererbung“ von Managern ist gleichzeitig deren Existenzberechtigung. Manager implementieren im Prinzip ein Verhalten, das eigentlich Bestandteil des POA ist. Durch die Verlagerung in Managerobjekte kann dieses Verhalten aber gleichzeitig für mehrere POAs genutzt werden.

In den folgenden Abschnitten wird die Funktionalität des POA anhand der Policies und zwei der wichtigsten Manager, dem *POA–Manager* und dem *Servant–Manager*, vorgestellt.

Policies

Die hier beschriebenen Policies regeln das Verhalten eines POA bezüglich der in Abschnitt 5.2.1 angesprochenen Funktionalität.

- **Threading Policy:** steuert die Parallelität der Ausführung von Methodenaufrufen durch Verwendung von mehreren Threads pro Prozeß. Im Moment werden zwei Modi unterstützt: Verwendung von nur einem Thread pro Prozeß (Single–Threading) oder Multi–Threading durch den ORB (d.h. der ORB und nicht der POA steuert die Verwendung von Threads).

- **Request Processing Policy:** steuert die Vorgehensweise bei einer Verkörperung. Es können drei Verhalten eingestellt werden. Erstens: Verwendung eines *Servant Manager*; zweitens: Verwendung eines Default–Servants und drittens: Abbruch des Methodenaufrufes, wenn das Zielobjekt gerade an keinen Servant gekoppelt ist.

- **Servant Retention Policy:** steuert die Durchführung von Vergeistigungen. Es kann entweder nach jedem Methodenaufruf eine Vergeistigung durchgeführt werden oder die Verbindung zwischen Objekt und Servant bleibt bestehen. Mittels dieser Policy kann beispielsweise der Aktivierungsmodus *Per–Method* des BOA nachgestellt werden.

- **Implicit Activation Policy:** steuert die Durchführung von Verkörperungen. Optional kann der POA bei Zugriff auf ein Objekt ohne Servant automatisch eine Verkörperung durchführen.

- **Lifespan Policy:** definiert die Art der vom POA verwalteten Objekte (transient oder persistent).

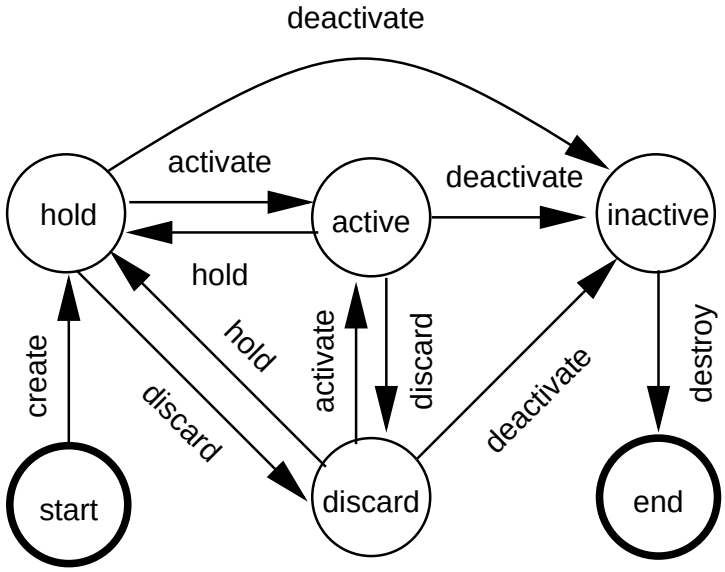


Abbildung 5.4: Zustandsdiagramm der POA–Manager

- **Object ID Uniqueness Policy:** steuert die Zuordnung von Servants zu Objekten (ein Servant pro Objekt oder ein Servant gleichzeitig für mehrere Objekte)
- **Object ID Assignment Policy:** steuert die Vergabe von Objekt–IDs (durch den POA oder durch den Benutzer)

Manager

POA–Manager steuern die Aktivierung und die Deaktivierung von Objekten. Ein POA–Manager beeinflußt dabei alle Objekte aller ihm zugeordneten POAs. Dazu realisiert ein POA Manager einen endlichen Automaten mit sechs Zuständen (siehe Abbildung 5.4):

- **active:** Objekte sind aktiv, Methodenaufrufe können durchgeführt werden.
- **hold:** Objekte sind inaktiv, eintreffende Methodenaufrufe werden solange verzögert, bis der *hold* Zustand verlassen wird.
- **discard:** Objekte sind inaktiv, eintreffende Methodenaufrufe werden mit einer Fehlermeldung abgebrochen.
- **inactive:** der Server wird gerade terminiert, alle Methodenaufrufe werden mit einer Fehlermeldung abgebrochen.
- **start, end:** Start– bzw. Endzustand

Servant–Manager sind Callback–Objekte, über die die zugeordneten POAs die Verkörperung und die Vergeistigung auslösen. Für die Servant Manager wird vom POA eine Schnittstelle bereitgestellt, die der Anwender zu implementieren hat. Immer dann, wenn eine Verkörperung oder Vergeistigung nötig wird, ruft der POA eine entsprechende Methode auf der vom Benutzer bereitgestellten Implementierung auf, die dann die Verkörperung bzw. Vergeistigung tatsächlich durchführt.

Bewertung

Es liegen bis jetzt nur sehr wenige Erfahrungen mit dem POA vor, da dessen Spezifikation sehr jung ist und noch diskutiert wird. Insbesondere ist es unklar, ob er sein Hauptziel erreicht und serverseitigen Kode in CORBA–Applikationen portabel zwischen verschiedenen CORBA–Produkten macht. Das wird auch sehr stark davon abhängen, ob die Hersteller in ihren Produkten den POA um proprietäre Funktionalität erweitern. Die Vergangenheit hat gezeigt, daß solche Erweiterungen von Herstellern häufig zur Erlangung von Wettbewerbsvorteilen benutzt werden. Potentielle Kandidaten für solche Erweiterungen sind:

- *Serveraktivierung:* Die Spezifikation sagt nichts über automatische Aktivierung von Servern, insbesondere über die dazu nötige Registrierung von Servern beim POA.
- *Multi–Threading:* Einige Hersteller unterstützen bereits jetzt Modelle, mit denen der Anwender sehr frei über die Verwendung von Threads entscheiden kann. Die POA–Spezifikation unterstützt im Moment nur zwei sehr eingeschränkte Threading Policies.

5.3.3 Library Object Adapter

Für diesen Typ von Objekt–Adapter existiert nach dem Wissen des Autors keine Spezifikation, er wird lediglich als ein Beispiel in einem einführenden Kapitel der CORBA–Spezifikation kurz umrissen.

Beim BOA wurde davon ausgegangen, daß die Servants in Form von separaten Programmen vorliegen, die vom BOA während einer Verkörperung gegebenenfalls gestartet werden. Ein Vorteil dieses Ansatzes ist, daß zu einem laufenden CORBA–System jederzeit neue Servants in Form von neuen Programmen hinzugefügt werden können. Diesen Vorteil erkauft man sich jedoch teuer mit dem Aufwand für die Kommunikation zwischen Klient– und Serverprozeß (beispielsweise mittels IIOP).

Beim *Library Object Adapter (LOA)* liegen Servants in Form von Bibliotheken vor, die direkt mittels des in Abschnitt 3.3.7 beschriebenen dynamischen Linkens in den Prozeß des Klienten geladen werden. Dadurch können Klienten mit den Objekt–Implementierungen über normale Methodenaufrufe kommunizieren. Der

LOA kombiniert also die Offenheit des BOA (Server können jederzeit hinzugefügt und entfernt werden) mit der Effizienz Prozeß–lokaler Kommunikation (Methodenaufrufe).

Bis auf den Unterschied, daß beim BOA Servants als Programme und beim LOA als Bibliotheken vorliegen, unterscheiden sich die beiden Objekt–Adapter kaum. Die Idee lag daher nahe, beim MICO–BOA einen neuen Aktivierungsmodus *Library* einzuführen, der Servants aus einer Bibliothek lädt.

5.3.4 Object–Oriented Database Object Adapter

Die Idee beim OODB OA ist, einem CORBA–System die in einer objektorientierten Datenbank abgelegten Objekte zugänglich zu machen. Natürlich läßt sich das auch mit einem Allzweck Objekt–Adapter wie dem BOA erreichen, indem man Stellvertreterobjekte erzeugt, deren Methoden direkt auf die Datenbank zugreifen. Nachteil dieses Ansatzes ist jedoch die schlechte Skalierbarkeit: Für jedes Objekt in der Datenbank wird auch ein Stellvertreterobjekt benötigt, was für große Mengen von Objekten aus Speicherplatzgründen unpraktikabel ist.

Daher verwendet man spezielle OODB Objekt–Adapter, bei denen keine Stellvertreterobjekte mehr benötigt werden, sondern direkt auf die Datenbank zugegriffen wird. Beispielsweise erlaubt das Orbix *Object Database Adapter Framework (ODAF)* [21] CORBA–Applikationen den Zugriff auf in objektorientierten Datenbanken abgelegt Objekte.

5.4 Design des MICO–BOA

Obwohl der POA dem BOA offensichtlich in Hinblick auf Funktionalität und Güte der Spezifikation überlegen ist, erfordert eine konforme CORBA–Implementierung im Moment noch die Bereitstellung eines BOA.

In diesem Abschnitt wird der Entwurf für einen vollständigen BOA auf Basis des in Kapitel 3 beschriebenen Mikrokern–ORB vorgestellt. Das hier Gesagte baut auf der in Abschnitt 5.3.1 beschriebenen Terminologie und Funktionalität des BOA auf. Zunächst wird ein Überblick über die grobe Struktur des Entwurfes gegeben, um dann auf die einzelnen Komponenten näher einzugehen. Zuvor jedoch noch einmal ein Überblick über die Schlüsseleigenschaften der BOA–Spezifikation:

- Servants liegen in Form von ausführbaren Programmen (sogegannten *Servern*) vor
- Registrierung von Servern beim BOA mittels eines Implementation Repository
- Verschiedene Aktivierungsmodi beeinflussen das Starten von Servern durch den BOA

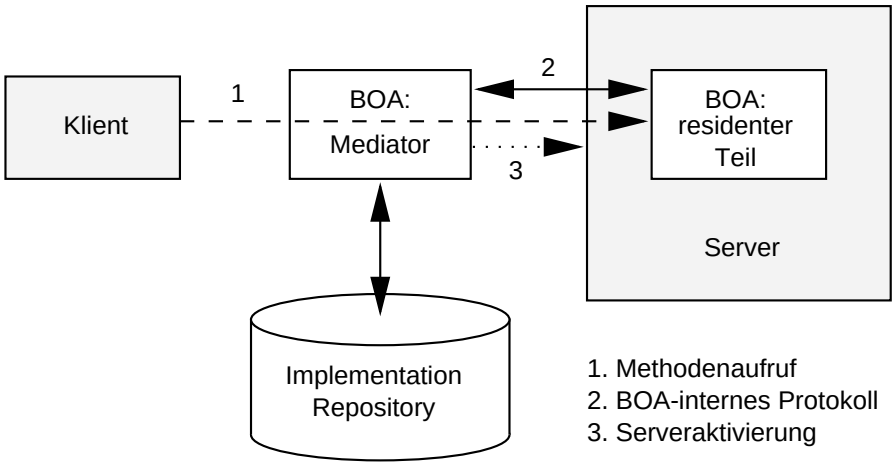


Abbildung 5.5: BOA–Komponenten

- CORBA–Objekte können während ihrer Lebenszeit durch verschiedene Servants aus verschiedenen Servern verkörpert werden

5.4.1 Überblick

Das Vorliegen der Servants als separate Programme, die durch den BOA gestartet werden müssen, impliziert, daß der BOA durch zwei separate Komponenten realisiert werden muß: einen *serverresidenten Teil* und einen *Mediator*. Wie in Abbildung 5.5 gezeigt, befindet sich der Mediator logisch zwischen Klient und Server und ist für das Starten der Serverprozesse verantwortlich, wobei er die für das Starten notwendigen Informationen dem Implementation Repository entnimmt. Außerdem verwaltet er die Identität der CORBA–Objekte, da diese über die Lebenszeit einzelner Servants (und damit auch Server) hinaus erhalten bleiben muß. Der serverresidente Teil befindet sich wie der Name schon sagt im Server und implementiert die restliche Funktionalität des BOA wie etwa die Ausführung von Methodenaufrufen. Neben der reinen Übertragung von Methodenaufrufen zwischen Mediator und Server ist ein umfangreiches Protokoll zwischen Mediator und serverresidentem Teil nötig. Über dieses Protokoll meldet sich beispielsweise ein Server beim Mediator, wenn er nach dem Starten seine Initialisierung abgeschlossen hat und bereit ist, Methodenaufrufe entgegenzunehmen und auszuführen.

Der BOA besteht also aus den drei in Abbildung 5.5 nicht schraffiert dargestellten Komponenten Implementation Repository, BOA–Mediator und serverresidenter BOA. Die nächsten Abschnitte sind diesen Komponenten gewidmet.

5.4.2 Implementation Repository

Die Informationen im Implementation Repository steuern die Aktivierung von Servern durch den Mediator. Obwohl das Implementation Repository damit ein

zentraler Teil des BOA ist, wird weder seine Struktur noch seine Schnittstelle durch die CORBA–Spezifikation vorgegeben.

Das MICO–Implementation Repository besteht aus zwei Komponenten, den *Server–Einträgen*, die alle für einen Server benötigten Informationen beinhalten, und dem *Repository* selbst, das zur Erzeugung und Verwaltung der Server–Einträge dient.

Server–Einträge

Ein Server–Eintrag enthält folgende Informationen über einen Server:

- eindeutiger Name
- bereitgestellte Servants
- Aktivierungsmodus
- Start–Kommando

Während einer Verkörperung muß einem CORBA–Objekt ein Servant zugeordnet werden. Der Mediator muß dann anhand der Informationen im Implementation Repository entscheiden können, welcher Server die zu dem Objekt gehörigen Servants implementiert. Im Implementation Repository muß also für jeden Server verzeichnet sein, welche Servants er bereitstellt. Zu diesem Zweck enthält ein Server–Eintrag die Typen der vom zugehörigen Server implementierten Objekte. Da zwei Server durchaus Objekte gleichen Typs auf verschiedene Art und Weise implementieren können, wird zusätzlich für jeden Eintrag ein eindeutiger Name vergeben. Daneben enthält jeder Eintrag neben dem Aktivierungsmodus einen Kommando–String, mit dem der zugehörige Server vom Mediator gestartet werden kann. Die zugehörige IDL–Schnittstelle sieht folgendermaßen aus:

```
// IDL
interface ImplementationDef {
    enum ActivationMode {
        ActivateShared,
        ActivateUnshared,
        ActivatePerMethod,
        ActivatePersistent
    };

    typedef sequence<string> RepoIdList;

    // Name
    readonly attribute string name;
```

```
        // Aktivierungsmodus
        attribute ActivationMode mode;

        // Liste implementierter Objekte
        attribute RepoIdList repoids;

        // Start-Kommando
        attribute string command;
    };
```

Die Objekttypen werden durch *Repository IDs* repräsentiert, spezielle Strings, mittels der die zugehörigen Typinformationen aus dem Interface Repository abgerufen werden können (siehe auch Abschnitt 2.5.1).

Repository

Das Repository dient als Factory zur Erzeugung und Zerstörung von Server–Einträgen im Implementation Repository und enthält zusätzlich Operationen zum Auffinden von Server–Einträgen anhand von gegebenen Namen oder Objekttypen.

```
    // IDL
    interface ImplRepository {
        typedef sequence<ImplementationDef> ImplDefSeq;

        // Erzeugung
        ImplementationDef create (
             in ImplementationDef::ActivationMode mode,
             in ImplementationDef::RepoIdList repoids,
             in string name, in string command);

        // Zerstoerung
        void destroy (in ImplementationDef impl_def);

        // Lookup
        ImplDefSeq find_by_name (in string name);
        ImplDefSeq find_by_repoid (in string repoid);
        ImplDefSeq find_all ();
    };
```

5.4.3 Mediator

Die in Abbildung 5.5 gezeigte Struktur läßt vermuten, daß für die Realisierung des Mediators ein erheblicher Aufwand nötig ist. Hier bewährt sich jedoch die

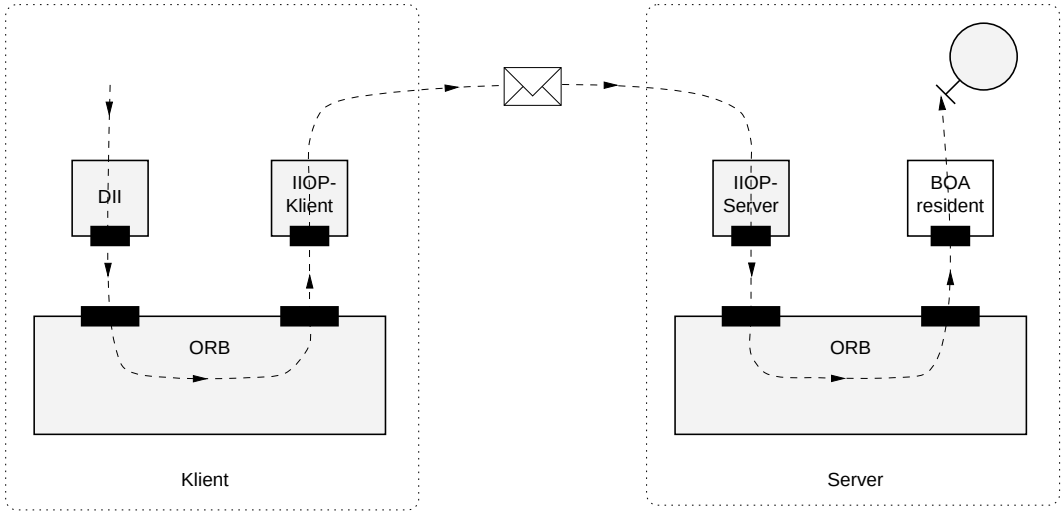


Abbildung 5.6: Direkte Kommunikation zwischen Klient und persistentem Server

in Kapitel 3 vorgestellte Mikrokern–ORB–Architektur, mit der sich der gesamte Mediator mittels eines CORBA–Objektes (genannt *Mediator–Objekt*) und eines speziellen Objekt–Adapters (genannt *Mediator–Adapter*) realisieren läßt.

Die Idee dafür ist folgende: Server mit dem Aktivierungsmodus *Persistent* werden vom Benutzer gestartet und müssen daher wie in Abschnitt 5.3.1 beschrieben nicht durch den Mediator aktiviert werden. Daher wird der Mediator für Objekte, die mittels persistenter Server realisiert werden, nicht benötigt. Klienten können also wie in Abbildung 5.6 gezeigt direkt mit persistenten Servern unter Umgehung des Mediators kommunizieren, beispielsweise mittels IIOP. Der serverresidente Teil des BOA wird so realisiert, daß er für persistente Server ohne Verwendung des Mediators auskommt, für die anderen Aktivierungsmodi jedoch den Mediator verwendet.

Der Mediator wird wie in Abbildung 5.7 gezeigt als separater Prozeß mit eigenem ORB realisiert, der zwischen Klient und Server geschaltet wird. Der Mediator verhält sich dem Klienten gegenüber wie ein Server, das heißt, ein Klient merkt nicht, daß er mit dem Mediator anstelle eines Servers kommuniziert. Ein vom Klient mittels IIOP abgeschickter Methodenaufruf erreicht den Mediator, dessen ORB den Methodenaufruf an den *Mediator–Adapter* weiterleitet. Falls für das Zielobjekt des Methodenaufrufes gerade kein Server aktiv ist, so aktiviert der Mediator–Adapter einen Server. Schließlich leitet der Mediator–Adapter den Methodenaufruf unter erneuter Verwendung seines ORB an den serverresidenten Teil des BOA im Server weiter. Dadurch ergibt sich der in Abbildung 5.7 gezeigte schleifenförmige Verlauf des Methodenaufrufes durch den Mediator.

Für das interne Protokoll zwischen Mediator und serverresidentem BOA wird wie in Abbildung 5.8 dargestellt ein CORBA–Objekt, das sogenannte *Mediator–Objekt*, verwendet. Das Mediator–Objekt wird über einen serverresidenten BOA

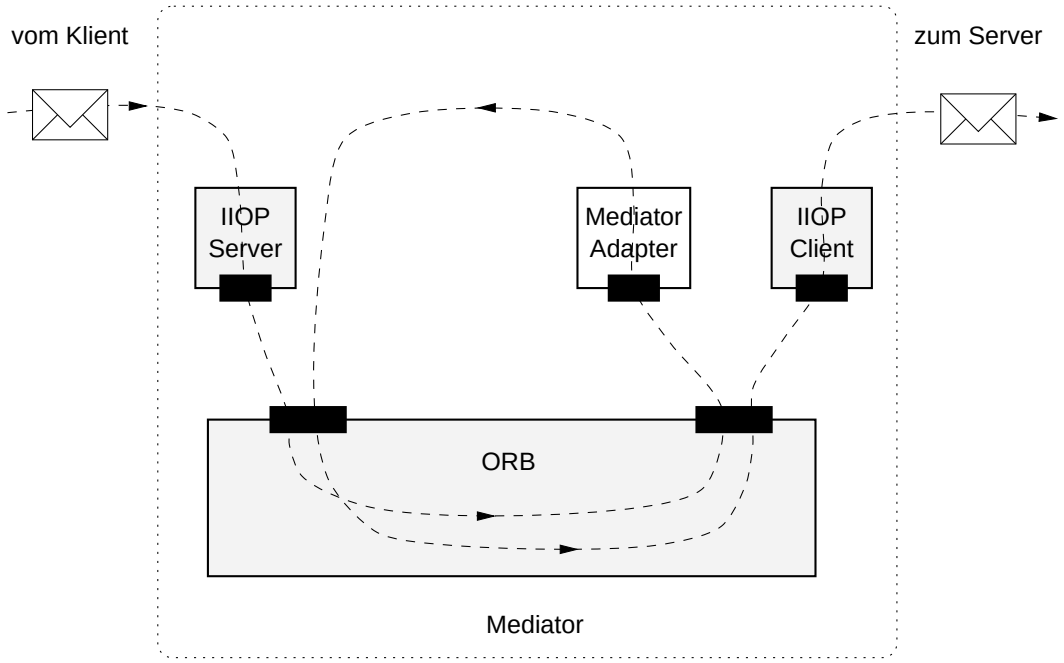


Abbildung 5.7: Methodenaufruf durch den BOA–Mediator

an den ORB gekoppelt, das heißt, der Mediator wird selbst zu einem Server gemacht. Das funktioniert, weil der serverresidente BOA wie oben beschrieben für persistente Server ohne den Mediator auskommt. Das Mediator–Objekt arbeitet eng mit dem Mediator–Adapter zusammen, indem es alle Methodenaufrufe an den Mediator–Adapter delegiert. In den nächsten beiden Abschnitten wird näher auf Aufbau und Funktionsweise von Mediator–Adapter und –Objekt eingegangen.

Mediator–Adapter

Der Mediator–Adapter stellt folgende Funktionalität zur Verfügung:

- Verwaltung der Identität von Objekten
- Aktivierung von Servern
- Weiterleitung von Methodenaufrufen an Server

Die Identität von CORBA–Objekten wird nach außen hin durch Objektreferenzen repräsentiert. Da CORBA–Objekte während ihrer Lebenszeit durch verschiedene Servants aus verschiedenen Servern verkörpert werden können, kann sich auch die Objektreferenz eines CORBA–Objektes ändern, da diese Informationen über den Aufenthaltsort des zugeordneten Servants enthält. Der Mediator hat die Aufgabe, diese Änderung der Objektreferenz vor den Klienten zu verbergen. Er realisiert

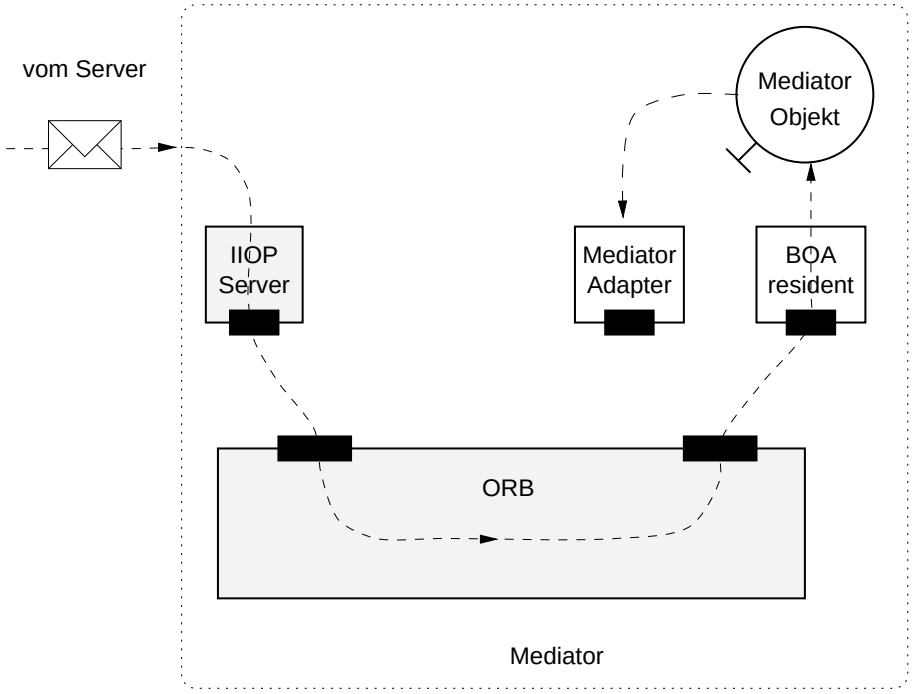


Abbildung 5.8: Kommunikation zwischen Mediator und serverresidentem BOA

damit eine sogenannte *Location–Domäne*, innerhalb der Migration und Evolution von Objekten transparent für Klienten möglich ist.

Um diese Veränderung der Objektreferenz vor den Klienten zu verbergen, generiert der Mediator für jedes Objekt eine sogenannte *Mediator–Objektreferenz*, mit der auf das zugehörige Objekt während seiner gesamten Lebenszeit zugegriffen werden kann—unabhängig davon, durch welchen Server das Objekt gerade verkörpert wird. Nur die Mediator–Objektreferenz wird Klienten bekanntgegeben. Jeder Server, durch den das Objekt während seiner Lebenszeit verkörpert wird, generiert eine eigene, sogenannte *Server–Objektreferenz* für das Objekt.

Kernstück des Mediators ist die sogenannte *Objekttabelle*, die für jedes vom Mediator verwaltete Objekt einen Eintrag enthält. Jeder Eintrag besteht aus folgenden vier Komponenten:

- Mediator–Objektreferenz
- Server–Objektreferenz
- Methodenwarteschlange
- Server–Zustand

Über diese Tabelle ordnet der Mediator jeder Mediator–Objektreferenz die im Moment gültige Server–Objektreferenz zu. Außerdem enthält jeder Eintrag eine Warteschlange, in der Methodenaufrufe auf dem Objekt zwischengespeichert

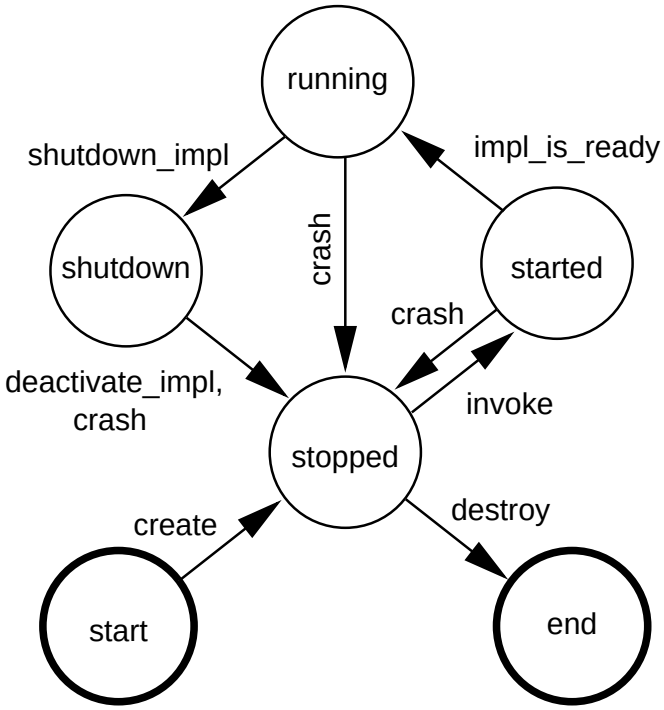


Abbildung 5.9: Zustandsdiagramm der Objekte im Mediator

werden, sowie Informationen über den momentanen Aktivierungszustand des zum Objekt zugeordneten Servers. Dieser wird durch einen endlichen Automaten mit den folgenden sechs Zuständen repräsentiert:

- **stopped:** dem Objekt ist gerade kein Server zugeordnet
- **started:** der dem Objekt zugeordnete Server wird gerade gestartet
- **running:** der dem Objekt zugeordnete Server ist bereit, Methodenaufrufe entgegenzunehmen und auszuführen
- **shutdown:** der dem Objekt zugeordnete Server wird gerade beendet
- **start, end:** Start– bzw. Endzustand

Im Zustand *running* werden beim Mediator eintreffende Methodenaufrufe direkt an den entsprechenden Server weitergeleitet. In allen anderen Zuständen außer *start* und *end* werden Methodenaufrufe solange verzögert, bis der Zustand *running* erreicht ist. Dafür wird die oben erwähnte Methodenwarteschlange benötigt. Abbildung 5.9 zeigt das Zustandsdiagramm des aus den oben genannten Zuständen gebildeten endlichen Automaten. Die Ereignisse, die darin einen Zustandsübergang auslösen, haben folgende Bedeutung:

- **create:** Erzeugung eines neuen CORBA–Objektes
- **destroy:** Zerstörung eines CORBA–Objektes
- **invoke:** Eintreffen eines Methodenaufrufes beim Mediator
- **crash:** Absturz des Servers
- **impl_is_ready:** Meldung vom Server, daß er bereit ist, Methodenaufrufe auszuführen
- **shutdown_impl:** Meldung vom Server, daß er ab sofort keine Methodenaufrufe mehr entgegennehmen kann
- **deactivate_impl:** Meldung vom Server, daß er sich beenden wird

Bis auf *invoke* und *crash* werden alle Ereignisse vom serverresidenten BOA über Methodenaufrufe auf dem in Abschnitt 5.4.3 beschriebenen Mediator–Objekt ausgelöst.

Trifft ein Methodenaufruf beim Mediator ein, sucht er zunächst den Eintrag in seiner Objekttabelle, dessen Mediator–Objektreferenz mit der Referenz des Zielobjektes des Methodenaufrufes übereinstimmt. Ist der zugeordnete Server im Zustand *running*, so wird der Methodenaufruf unter Verwendung der Server–Objektreferenz aus der Tabelle direkt zum Server weitergeschickt. Im Zustand *stopped* konsultiert der Mediator das Implementation Repository, um einen geeigneten Server zu finden und zu starten. Außerdem wird der Methodenaufruf in die Warteschlange eingefügt. Sobald der Server den Zustand *running* erreicht, werden die zwischengespeicherten Methodenaufrufe an den Server weitergeschickt.

Mediator–Objekt

Mittels des Mediator–Objektes wird die Kommunikation zwischen Mediator und serverresidentem BOA realisiert. Die Implementierung der unten gezeigten Schnittstelle des Mediator–Objektes enthält selbst keine Funktionalität, sondern delegiert die Methodenaufrufe an den Mediator–Adapter.

```
// IDL
interface BOAMediatorObject {

    // Erzeuge neues Objekt
    void create (in Object server_objref,
                 in ImplementationDef impl,
                 out Object mediator_objref,
                 ...);
```

```
        // Zerstoere Objekt
        void dispose (in Object mediator_objref);

        // Erfrage zu verkoerpernde Objekte beim Mediator
        sequence<Object> get_restore_objs (
                in ImplementationDef impl);

        // Verkoerpere existierendes Objekt
        void restore (in Object server_objref,
                      inout Object mediator_objref,
                      ...);

        // Objekt Migration/Evolution
        void change_impl (in Object mediator_objref,
                          in ImplementationDef impl);

        // Synchronisation Server/Mediator fuer
        // SHARED Server
        void impl_is_ready (in ImplementationDef impl);
        void shutdown_impl (in ImplementationDef impl);
        void deactivate_impl (in ImplementationDef impl);

        // Synchronisation Server/Mediator fuer
        // UNSHARED und PERMETHOD Server
        void obj_is_ready (in Object objref,
                           in ImplementationDef impl);
        void shutdown_obj (in Object objref);
        void deactivate_obj (in Object objref,
                             in ImplementationDef impl);
    };
```

Die Operationen der Schnittstelle `BOAMediatorObject` können in vier Kategorien eingeteilt werden:

- Erzeugung/Zerstörung von Objekten
- Verkörperung von Objekten
- Evolution/Migration von Objekten
- Synchronisation zwischen Server und Mediator

Bei der Erzeugung eines Objektes mittels `create` wird wie in Abschnitt 5.4.3 beschrieben die Server–Objektreferenz an den Mediator übertragen, der eine neue

Mediator–Objektreferenz generiert, die als Ausgabeparameter dem Aufrufer mitgeteilt wird. Die Verkörperung wird in zwei Schritten vorgenommen. Mittels `get_restore_objs` werden die Mediator–Objektreferenzen der zu verkörpernden Objekte beim Mediator erfragt. Für jedes dieser Objekte führt der serverresidente BOA die Verkörperung durch, generiert eine neue Server–Objektreferenz und teilt diese mittels `restore` dem Mediator mit. Mittels `change_impl` kann einem Objekt ein anderer Server (spezifiziert durch einen `ImplementationDef`–Eintrag im Implementation Repository) zugeordnet werden. Damit wird die Evolution bzw. Migration von Objekten zur Laufzeit ermöglicht. Die Operationen zur Koordination zwischen Server und Mediator bewirken einen Zustandsübergang des im Abschnitt 5.4.3 beschriebenen Zustandsautomaten. Je nach Aktivierungsmodus des Servers werden die Operationen mit `impl` (Aktivierungsmodus *shared*) bzw. `obj` (Aktivierungsmodi *unshared* und *per–method*) im Namen verwendet.

5.4.4 Serverresidenter BOA

Wie schon in Abschnitt 5.4.3 angedeutet, wird der serverresidente BOA zweistufig realisiert. Der Kern kommt ohne Mediator aus und enthält all die Funktionalität, die für persistente Server benötigt wird. Dieser Kern wird um Funktionalität erweitert, die den Mediator zur Realisierung der anderen Aktivierungsmodi verwendet. Zu den Aufgaben des Kerns zählt:

- Generierung von Server–Objektreferenzen
- Aktivierung/Deaktivierung von Objekten
- Verkörperung/Vergeistigung von Objekten
- Durchführung von Methodenaufrufen

Der Kern unterstützt nur transiente Objekte und damit ein stark vereinfachtes Modell für Aktivierung/Deaktivierung und Verkörperung/Vergeistigung: Bei der Erzeugung eines Objektes wird automatisch eine Aktivierung und Verkörperung durchgeführt. Die Zerstörung eines Objektes impliziert eine Vergeistigung und Deaktivierung. Objekte bleiben während ihrer gesamten Lebenszeit aktiv und verkörpert. Erst die Erweiterung des Kerns und damit die Verwendung des Mediators ermöglicht das in Abschnitt 5.2.1 beschriebene flexiblere Modell für Aktivierung/Deaktivierung und Verkörperung/Vergeistigung von Objekten. Die Aufgabe der Erweiterung besteht in der Kooperation mit dem Mediator durch Methodenaufrufe auf dem in Abschnitt 5.4.3 beschriebenen Mediator–Objekt.

BOA–Kern

Die Hauptaufgabe des BOA–Kerns besteht in der Durchführung von Methodenaufrufen. Zentrale Komponente des Kerns ist eine Objekttabelle, mittels der Ob-

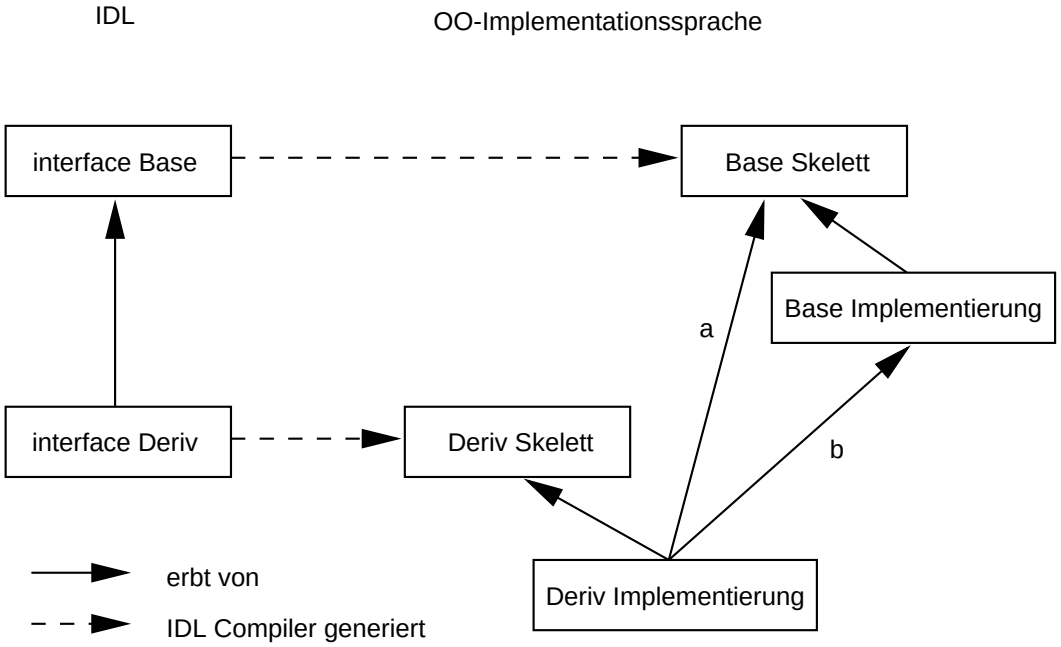


Abbildung 5.10: Kode– vs. Schnittstellenvererbung

jektreferenzen Servants zugeordnet werden. Beim Eintreffen eines Methodenaufrufes bestimmt der serverresidente BOA wie in Abschnitt 5.2.2 beschrieben mittels der Objekttabelle den zum Zielobjekt gehörigen Servant. Dieser bestimmt das für das Zielobjekt zuständige Skelett und überläßt diesem die weitere Ausführung des Methodenaufrufs.

Die Anbindung von durch den Anwendungsprogrammierer bereitgestellten Methodenimplementationen an statische Skelette erfolgt in objektorientierten Programmiersprachen durch Vererbung. Dabei ist das Skelett eine Basisklasse, von der der Programmierer seine Objekt–Implementierung ableitet und dabei die Methoden implementiert.

Bei der Anbindung von bestehenden Systemen an CORBA liegen die Methodenimplementationen jedoch oftmals schon in Form einer kommerziellen Klassenbibliothek vor, für die kein Quellkode verfügbar ist.

In diesem Fall werden spezielle Objekt–Implementierungen (sogenannte *TIEs*) verwendet, die einfach alle Methodenaufrufe an die Bibliothek delegieren. In Sprachen mit generischen Datentypen (wie beispielsweise Templates in C++) können TIEs automatisch vom IDL–Compiler generiert werden.

Während Vererbung in IDL nur die Vererbung und damit Wiederverwendung von *Schnittstellen* bedeutet (siehe Abschnitt 2.5.1), ist es oftmals jedoch wünschenswert, auch die *Implementierungen* von Methoden der Basisschnittstelle zu erben und damit wiederzuverwenden. Während das im allgemeinen Fall einen erheblichen Aufwand erfordert (siehe [18]), ist der Spezialfall, bei dem sich die Implementierungen von Basisklasse(n) und abgeleiterer Klasse im gleichen

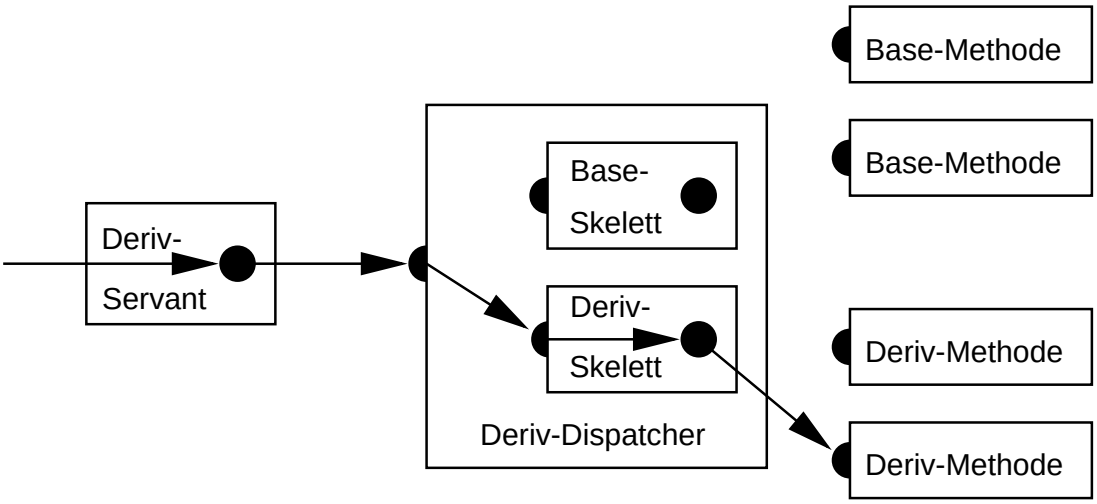


Abbildung 5.11: Dispatching von Methodenaufrufen im BOA

Server befinden, einfacher zu handhaben.

In MICO lassen Skelette dem Anwender die Wahl zwischen Schnittstellen– und Kode–Vererbung. Wie in Abbildung 5.10 dargestellt, werden dazu die über eine IDL–Vererbungsbeziehung verknüpften Schnittstellen *Base* und *Deriv* vom IDL–Compiler auf je eine Skelett–Klasse in der Implementationssprache abgebildet. Der Anwendungsprogrammierer stellt eine Implementierung für die Schnittstelle *Base* bereit, indem er eine Klasse *Base–Implementierung* von *Base–Skelett* ableitet und die Methoden der Schnittstelle implementiert. Für die Implementierung von *Deriv* hat er nun die Wahl, ob er nur die Schnittstelle von *Base* oder auch deren Implementierung erben möchte. Im ersten Fall erbt er von *Base–Skelett* (Pfeil „a“ in Abbildung 5.10) und *Deriv–Skelett* und muß sowohl die Methoden von *Base* als auch von *Deriv* implementieren. Im zweiten Fall erbt er von *Base–Implementierung* (Pfeil „b“ in Abbildung 5.10) und *Deriv–Skelett*, erbt somit die Implementierung der Methoden von *Base* und muß nur noch die Methoden von *Deriv* implementieren. So kann der Programmierer mit MICO für jede Basisklasse separat entscheiden, ob er nur deren Schnittstelle oder auch die Implementierung erben möchte—vorausgesetzt die Implementierungen der Basisklassen befinden sich im gleichen Server.

Wie in Abschnitt 5.2.2 beschrieben, werden Methodenaufrufe durch einen mehrstufigen Dispatch–Vorgang an die Methodenimplementierung weitergeleitet. Die letzte Stufe diese Vorgangs bilden die Skelette (Pfeil 3 in Abbildung 5.2). Bedingt dadurch, daß bei Verwendung von Vererbung mehrere Skelette für ein CORBA–Objekt zuständig sind, ist dieser letzte Schritt im MICO–BOA zweistufig. Wie in Abbildung 5.11 gezeigt wird dazu für jedes CORBA–Objekt ein zusätzlicher *Dispatcher* benötigt, der die Skelette aller am CORBA–Objekt durch Vererbung beteiligten Schnittstellen enthält. Trifft ein Methodenaufruf beim Dis-

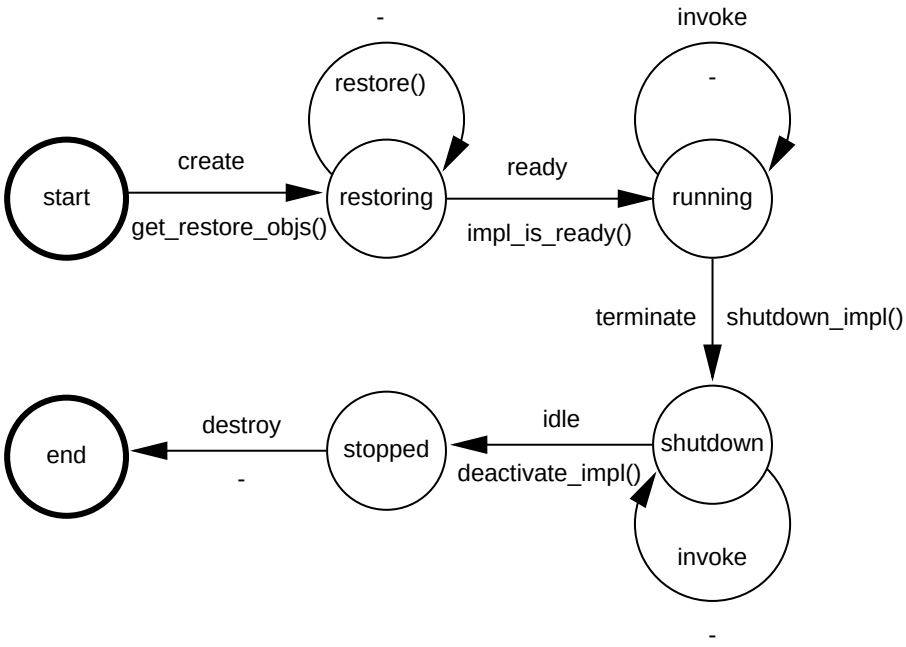


Abbildung 5.12: Lebenszyklus eines BOA–Servers

patcher ein, so ist er dafür zuständig, das Skelett auszuwählen, dessen Schnittstelle die aufgerufene Methode enthält. Dieses Skelett wählt dann die Methodenimplementierung aus und führt den Methodenaufruf schließlich durch.

Kooperation mit dem Mediator

Der Lebenszyklus eines Servers wird durch einen Zustandsautomaten repräsentiert, der eng mit dem in Abschnitt 5.4.3 beschriebenen Automaten im Mediator–Adapter gekoppelt ist. Die Kopplung der beiden Automaten erfolgt über Methodenaufrufe auf dem im Abschnitt 5.4.3 beschriebenden Mediator–Objekt.

Abbildung 5.12 zeigt das Zustandsdiagramm des erweiterten endlichen Automaten, der den Lebenszyklus eines BOA–Servers repräsentiert. Darin sind über bzw. links der Übergangspfeile die Ereignisse benannt, die einen Übergang auslösen, darunter bzw. rechts die Aktionen, die der Automat bei einem Übergang auslöst. Die Aktionen sind dabei Methodenaufrufe auf dem Mediator–Objekt. Die Zustände des Automaten haben folgende Bedeutung:

- **restoring:** Server verkörpert Objekte (erzeugt Servants und koppelt diese an Objekte)
- **running:** Server ist bereit, Methodenaufrufe entgegenzunehmen und auszuführen
- **shutdown:** Server terminiert gerade
- **stopped:** Terminierung ist abgeschlossen
- **start, end:** Start– bzw. Endzustand

Folgende Ereignisse bewirken Zustandsübergänge im Automaten:

- **create:** Start eines Servers durch den Mediator
- **ready:** Server hat Initialisierung abgeschlossen
- **invoke:** Eintreffen eines Methodenaufrufes
- **terminate:** Server soll terminiert werden
- **idle:** alle noch ausstehenden Methodenaufrufe sind bearbeitet worden
- **destroy:** Beendigung des Servers

Nach seinem Start erfragt der Server beim Mediator mittels `get_restore_objs()` die Objektreferenzen der durch ihn zu verkörpernden Objekte und wechselt in den Zustand *restoring*. In diesem Zustand generiert er für jedes zu verkörpernde Objekt eine Server–Objektreferenz (siehe dazu auch Abschnitt 5.4.3) und einen Servant. Mittels `restore()` teilt er dem Mediator die so generierte Server–Objektreferenz mit. Ist die Verkörperung aller Objekte abgeschlossen, wechselt der Server in den Zustand *running* und teilt das dem Mediator durch Aufruf von `impl_is_ready()` mit, wodurch sich der Automat im Mediator ebenfalls in den Zustand *running* begibt. Nun ist der Server bereit, Methodenaufrufe entgegenzunehmen und auszuführen.

Nach Durchführung einer Vergeistigung werden die in einem Server–Prozeß ausgeführten Servants nicht mehr benötigt und der Server–Prozeß kann beendet werden. Dabei muß sichergestellt werden, daß keine Methodenaufrufe „verlorengehen“, die sich gerade auf dem Weg zwischen Mediator und Server–Prozeß befinden. Mit „auf dem Weg“ sind die diversen Puffer gemeint, in denen auf einer Netzwerkverbindung Daten zwischengespeichert werden:

- Sendepuffer beim Mediator
- Puffer von Routern

- Empfangspuffer beim Server

Das Ereignis *terminate* veranlaßt die Terminierung. Der Server ruft daraufhin die Methode `shutdown_impl()` auf dem Mediator–Objekt auf und begibt sich in den Zustand *shutdown*. Der Methodenaufruf bewirkt, daß sich der Automat im Mediator ebenfalls in den Zustand *shutdown* begibt und neu eintreffende Methodenaufrufe in seiner Warteschlange zwischenspeichert. Nun muß nur noch dafür gesorgt werden, daß vom Mediator bereits abgeschickte Methodenaufrufe noch ausgewertet werden. Dazu bleibt der Server im Zustand *shutdown* und führt eintreffende Methodenaufrufe (Ereignis *invoke*) aus. Treffen keine neuen Methodenaufrufe mehr ein (Ereignis *idle*), so begibt sich der Server in den Zustand *stopped* und ruft die Methode `deactivate_impl()` auf dem Mediator–Objekt auf, um dem Mediator anzuzeigen, daß die Terminierung abgeschlossen ist.

Das Ereignis *idle* stellt ein Problem dar, denn der Server kann nicht wissen, wann alle Puffer geleert sind und keine weiteren Methodenaufrufe mehr eintreffen können. Da die Kommunikation zwischen Mediator und Server über einen Kanal mit FIFO–Ordnung erfolgt[1], genügt es aber, wenn der Mediator beim Aufruf von `shutdown_impl()` eine spezielle Terminierungs–Nachricht an den Server schickt. Das Eintreffen dieser Nachricht beim Server ist dann gleichbedeutend mit dem Ereignis *idle*.

5.5 Zusammenfassung, Bewertung und Alternativen

Der Basic Object Adapter (BOA) ist der einzige im Moment von der OMG als Teil einer offiziellen Spezifikation definierte Objekt–Adapter. Jede standardkonforme CORBA–Implementierung muß einen BOA zur Verfügung stellen. Durch erhebliche Mängel in der Spezifikation unterscheiden sich BOA–Implementierungen verschiedener Hersteller erheblich in der vom Benutzer sichtbaren Schnittstelle und deren Semantik, so daß CORBA–Applikationen, die den BOA verwenden, nicht portabel sind.

Die zentrale Idee beim Entwurf des MICO–BOA ist die Verwendung eines Mediators, der für die Aktivierung von Servern und die Verwaltung von Objekt–Identitäten verantwortlich ist. Der gewählte Ansatz weist folgende Eigenschaften auf:

+ geringer Aufwand für Realisierung durch Ausnutzung der Mechanismen des Mikrokern–ORB

+ Unterstützung für Objektmigration und –evolution

[1] Das Vertauschen von Methodenaufrufen auf einem Objekt hätte fatale Folgen.

+ Unterstützung transienter und persistenter Objekte
+ Unterstützung aller Aktivierungsmodi
– Mediator ist Performanzengpaß und zentraler Ausfallpunkt
– hoher Kommunikationsaufwand

Der Performanzengpaß läßt sich relativieren, indem man mehrere Mediatoren verwendet. Jeder Mediator realisiert jedoch ein *Location–Domäne*, innerhalb der die Migration und Evolution von Objekten transparent für Klienten möglich ist. Da Migration zwischen Domänen nicht möglich ist, schränkt die Verwendung mehrerer Mediatoren die Freiheit der Migration ein.

Der erhöhte Kommunikationsaufwand resultiert daraus, daß zwei Netzwerktransfers (Klient → Mediator und Mediator → Server) notwendig sind.

Motivation für die Verwendung eines Mediators ist, daß Objekte während ihrer gesamten Lebenszeit unter ein und derselben Objektreferenz ansprechbar sein müssen. Durch Migration kann sich aber die Objektreferenz ändern, da diese Informationen über den Aufenthaltsort eines Objektes enthält. Diese Änderung der Objektreferenz verbirgt der Mediator vor den Klienten.

Will man ohne die Verwendung eines Mediators auskommen, so muß jedes migrierte Objekt an seinem ursprünglichen Aufenthaltsort einen Stellvertreter zurücklassen, weil Klienten noch im Besitz von Objektreferenzen sein können, die auf den ursprünglichen Aufenthaltsort des Objektes verweisen[2]. Das Stellvertreterobjekt muß dann dafür sorgen, daß Methodenaufrufe auf diesen Objektreferenzen an den neuen Aufenthaltsort des migrierten Objektes weitergeleitet werden. Migriert ein Objekt häufig, so sammeln sich mit der Zeit viele Stellvertreterobjekte an.

Aus dieser kurzen Diskussion wird deutlich, daß mit den Mitteln der CORBA–Spezifikation die Migration von Objekten mit einem Tradeoff zwischen Performanz und Ressourcenbedarf verbunden ist und daher schlecht skaliert. Einige mögliche Ansätze zur Verbesserung dieser Situation sind:

- Loslösung von Adreßinformationen aus Objektreferenzen und Bereitstellung eines Dienstes zur Abbildung von Objektreferenzen auf Adreßinformationen. Durch geeignete Caching–Strategien ist es möglich, diesen Dienst nur beim ersten Zugriff auf ein Objekt und beim jeweils ersten Zugriff auf ein Objekt nach einer Migration kontaktieren zu müssen.
- Bereitstellung eines *Push*–Mechanismus, mit dem Server aktiv alle im Umlauf befindlichen Objektreferenzen ändern können.

[2]Im Gegensatz zu DCOM haben Objekt–Implementierungen in CORBA keinen Überblick darüber, ob Klienten noch im Besitz von Objektreferenzen sind, die auf sie verweisen.

Die OMG diskutiert im Moment ein Modell [38], um CORBA–Objekte als Parameter von Methodenaufrufen *by–value* anstelle wie bisher *by–reference* übergeben zu können. Die Diskussion dazu verläuft aber parallel zu den mit der Migration verbundenen Problemen, da *by–value* übergebene Objekte eine vom Original–Objekt unabhängige Kopie erzeugen.

Kapitel 6

Aufruf–Adapter

Aufruf–Adapter bilden die Verbindung zwischen Aufrufer und ORB, ähnlich wie Objekt–Adapter die Verbindung zwischen ORB und Objekt–Implementierungen herstellen. Im Mikrokern–ORB sind Aufruf–Adapter vom ORB–Kern getrennte Services. In Analogie zu den Objekt–Adaptern verbergen sie verschiedene Typen von Aufrufern vor dem ORB und tragen damit dazu bei, den ORB–Kern klein und flexibel zu halten.

In diesem Kapitel werden zunächst die den Aufruf–Adaptern zugrundeliegenden Konzepte vorgestellt, um dann auf einen speziellen Adapter, das *Dynamic Invocation Interface (DII)*, und dessen Realisierung in MICO einzugehen.

6.1 Funktionalität

Aufruf–Adapter sind Komponenten des CORBA–Systems, die Klienten das Durchführen von Methodenaufrufen ermöglichen. Sie werden einerseits von Stub–Objekten benutzt, andererseits können sie direkt vom Benutzer zum Aufruf von Methoden auf Objekten verwendet werden, für die keine Stubs vorhanden sind.

Zur Durchführung eines Methodenaufrufes benötigen Aufruf–Adapter die Objektreferenz des Zielobjektes, den Methodennamen und Werte für die Eingabeparameter. Als Ergebnis eines Methodenaufrufes werden im Erfolgsfall der Ergebniswert und die Werte der Ausgabeparameter geliefert. Ist ein Fehler aufgetreten, wird statt dessen eine Exception zurückgeliefert.

Aufruf–Adapter können ein oder mehrere der CORBA–Aufrufsemantiken unterstützen (siehe auch Abschnitt 2.5.1):

- *synchron* mit *at–most–once* Semantik
- *asynchron* mit *at–most–once* Semantik
- *oneway* mit *best-effort* Semantik

Eigenschaft	statisch	dynamisch
Performanz	hoch	gering
Handhabung	einfach	kompliziert
Typinfos zur Compilierzeit	benötigt	nicht benötigt

Tabelle 6.1: orthogonale Eigenschaften dynamischer und statischer Aufruf–Adapter

CORBA unterscheidet *statische* und *dynamische* Aufruf–Adapter. Vom IDL–Compiler generierte Stub–Objekte verwenden statische Aufruf–Adapter. Weil letztere nicht direkt, sondern nur indirekt über Stub–Objekte vom Benutzer verwendet werden, ist das Design statischer Aufruf–Adapter nicht von der CORBA–Spezifikation vorgegeben. Dynamische Aufruf–Adapter und insbesondere das durch CORBA spezifizierte *Dynamic Invocation Interface (DII)* erlauben es, Methoden auf Objekten aufzurufen, für die keine Stubs vorhanden sind.

Tabelle 6.1 zeigt die Eigenschaften statischer und dynamischer Aufruf–Adapter im Vergleich. Die Handhabung statischer Aufruf–Adapter ist deshalb einfacher, weil sie nicht direkt, sondern nur über Stub–Objekte verwendet werden. Die Unterschiede in der Performanz resultieren einerseits daraus, daß das nicht vorgegebene Design statischer Aufruf–Adapter viel Raum für Optimierungen läßt. Andererseits kann der IDL–Compiler während des Übersetzens schon Aufgaben durchführen, die bei Verwendung dynamischer Aufruf–Adapter zur Laufzeit geschehen müssen (vergleiche auch Abschnitt 5.2.2).

Kern jedes Aufruf–Adapters ist ein Mechanismus zur Repräsentation von Wert und Typ beliebiger IDL–Datentypen. Insbesondere zusammengesetzte Datentypen wie Strukturen oder Unions sind bestimmend für die Komplexität eines solchen Mechanismus. Eine weitere wichtige Funktion von Aufruf–Adaptern ist die Typüberprüfung. Die nächsten beiden Abschnitte gehen auf diese Themen näher ein.

6.1.1 Repräsentation von IDL–Datentypen

CORBA spezifiziert als Teil des DII einen Standardmechanismus zur Repräsentation von Typ und Wert beliebiger IDL–Datentypen. Typen werden durch sogenannte `TypeCode`s zweistufig repräsentiert: Für die Basisdatentypen wie `long` existieren `TypeCode`–Konstanten. Beispielsweise wird der Typ `long` im C++–Mapping durch die Konstante `_tc_long` repäsentiert. Für zusammengesetzte Datentypen können `TypeCode`s konstruiert werden, die `TypeCode`s ihrer Elementdatentypen und weitere typbestimmende Daten, wie zum Beispiel die Indexgrenzen von Arrays, enthalten. Außerdem stellt CORBA einen Mechanismus bereit, um verschiedene `TypeCode`s auf Typgleichheit zu überprüfen.

Zur Darstellung von Werten sieht CORBA sogenannte `Any`–Objekte vor. Ein `Any`–Objekt enthält neben einem `TypeCode`, der den Typ des gerade im `Any` ent-

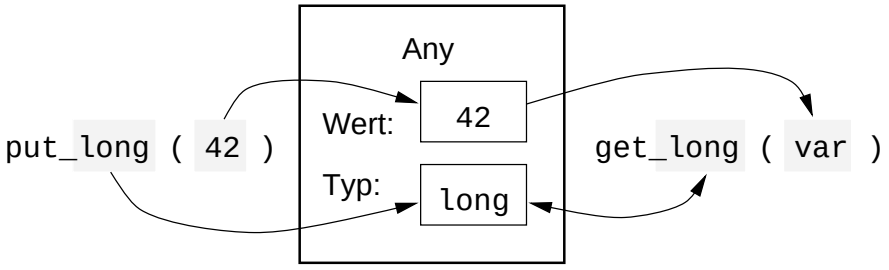


Abbildung 6.1: Einfügen und Auslesen von Daten in bzw. aus `Any`

haltenen IDL–Datentyps repräsentiert, einen nicht näher spezifizierten Mechanismus zum Vorhalten von Werten dieses Typs. `Any` verfügt an seiner Schnittstelle über Operationen zum typsicheren Einfügen und Auslesen aller Basisdatentypen:

```
// IDL
interface Any {
    // Einfuege-Operationen
    void put_long (in long l);
    void put_short (in short s);
    ...

    // Auslese-Operationen
    boolean get_long (out long l);
    boolean get_short (out short s);
    ...
};
```

„Typsicher“ bedeutet hier, daß die Einfüge–Operationen automatisch den im `Any` enthaltenen `TypeCode` setzen und die Auslese–Operationen den Typ des im `Any` gespeicherten Wertes mit dem der Variable vergleichen, in die der Wert abgelegt werden soll. Wie in Abbildung 6.1 gezeigt, ist die Typinformation dabei implizit in der verwendeten Einfüge– bzw. Auslese–Operation enthalten.

6.1.2 Typüberprüfung

Die Typüberprüfung sorgt dafür, daß Anzahl, Typ und Wert von Parametern eines Methodenaufrufes kompatibel zur Signatur der aufgerufenen Methode sind. Der Aufwand für die Typüberprüfung hängt stark ab von:

- Art des Aufruf–Adapters (statisch oder dynamisch)
- Typisierung der Implementationssprache (statisch oder dynamisch)

Am einfachsten ist die Typüberprüfung bei Verwendung statischer Aufruf–Adapter in statisch typisierten Programmiersprachen wie beispielsweise C++. Da statische Aufruf–Adapter nie direkt, sondern nur indirekt über Stub–Objekte verwendet werden, wird die Typüberprüfung vollständig vom Compiler oder Interpreter der Implementationssprache übernommen.

Dagegen müssen dynamische Aufruf–Adapter und statische Aufruf–Adapter in nicht statisch typisierten Implementationssprachen die Typüberprüfung selbst übernehmen. Dabei kommt das Interface Repository zur Anwendung, in dem die Signaturen der Methoden in Form einer Menge von `TypeCodes` abgelegt sind. Anhand der Objektreferenz des Zielobjektes und des Methodennamens kann der Aufruf–Adapter diese `TypeCodes` abrufen und mit den Typen der übergebenen Parameter vergleichen.

6.2 Dynamic Invocation Interface

Das *Dynamic Invocation Interface (DII)* ist ein durch CORBA spezifizierter dynamischer Aufruf–Adapter. Mit seiner Hilfe können Methoden auf Objekten aufgerufen werden, für die keine Stubs vorhanden sind.

Zentrale Komponente des DII sind sogenannte `Request`–Objekte, die einen Methodenaufruf repräsentieren und Name der aufzurufenden Methode, Objektreferenz des Zielobjektes und Parameter enthalten.

Um einen Methodenaufruf über das DII abzuwicklen, werden alle Parameter des Methodenaufrufes in `Anys` verpackt und zusammen mit der Objektreferenz des Zielobjektes und dem Methodennamen an das DII übergeben. Als Beispiel betrachte man die Schnittstelle `Konto`, die ein Bankkonto modelliert und über Operationen zum Einzahlen, Abheben und Anzeigen des Kontostands verfügt:

```
// IDL
interface Konto {
    void einzahlen (in long betrag);
    void abheben (in long betrag);
    long kontostand ();
};
```

Das folgende C++ Kode–Stück zeigt den Aufruf der Methode `einzahlen(100)` unter Verwendung des DII:

```
// C++
// Objektreferenz fuer Konto-Objekt besorgen
CORBA::Objekt_var zielobjekt = ...;
```

```
// DII Request erzeugen
CORBA::Request_var dii_request =
    zielobjekt->_request ("einzahlen");

// Argument 'betrag' angeben
dii_request->add_in_arg ("betrag") <<= (CORBA::Long)100;

// Methode aufrufen
dii_request->invoke ();
```

Der Aufruf von `_request()` auf dem Zielobjekt des Methodenaufrufes mit dem Namen der auzurufenden Methode als Parameter erzeugt ein neues `Request`–Objekt, das den DII–Methodenaufruf repräsentiert. Mittels `add_in_arg()` wird der Eingabeparameter `betrag` zur Liste der Parameter hinzugefügt. Der Aufruf liefert eine Referenz auf ein `Any`–Objekt zurück, in das mittels des überladenen Operators `<<=`[1] der Wert 100 eingetragen wird. Der Aufruf von `invoke()` übergibt den Methodenaufruf an den ORB und blockiert dann so lange, bis der Methodenaufruf abgearbeitet ist.

6.3 Subtyping

Im Moment vollzieht sich eine Wandlung der Umgebungen, in denen CORBA–Systeme eingesetzt werden weg von abgeschlossenen Systemen hin zu *offenen Dienstmärkten*, in denen Dienste in Form von CORBA–Objekten frei angeboten und nachgefragt werden. Einige Ursachen für diese Wandlung sind:

- Spezifikation von Mechanismen, die Interoperabilität zwischen CORBA–Produkten verschiedener Hersteller ermöglichen (siehe dazu Kapitel 4)
- Popularität von CORBA, beispielsweise enthält der Netscape Communicator 4 einen CORBA–ORB [33]

Offene Dienstmärkte zeichnen sich dadurch aus, daß Dienstnutzer keine Vorab–Kenntnisse über die angebotenen Dienste und umgekehrt haben. Um den Anforderungen eines solchen offenen Dienstmarktes gerecht zu werden, müssen Klienten in einem CORBA–System dazu fähig sein, mit Diensten zusammenzuarbeiten, über die sie keine *a priori* Kenntnisse besitzen. Genau das ist aber nicht der Fall, denn Voraussetzung für die Zusammenarbeit eines Klienten mit einem Dienst ist, daß die vom Klient erwartete Schnittstelle und die vom Dienst tatsächlich bereitgestellte Schnittstelle in einer Vererbungsbeziehung stehen. Dienstnutzer und

[1] Im C++–Mapping werden die Einfüge– und Auslese–Operationen auf `Any`–Objekten auf `<<=` bzw. `>>=` Operatoren abgebildet.

Dienstanbieter müssen also A–priori–Kenntnis voneinander haben, und zwar in Form einer IDL–Schnittstelle.

Zwar läßt die Verwendung des DII die Konstruktion generischer Klienten [48] zu, die nicht über diese Einschränkung verfügen, doch dadurch geht die Eleganz und einfache Handhabung der Stubs für den Anwender verloren. Außerdem benutzt im Moment nur ein verschwindend geringer Teil der existierenden CORBA–Applikationen das DII, so daß ein erheblicher Aufwand für die Umstellung dieser Applikation notwendig wäre.

Eine Möglichkeit die Notwendigkeit von Vorab–Kenntnissen zwischen Dienstnutzern und –anbietern abzuschwächen, ohne bestehende CORBA–Applikationen ändern zu müssen, ist die Ersetzung der statischen IDL–Vererbungsbeziehung durch eine dynamisch überprüfbare Subtyp–Relation [3] zwischen Schnittstelle des angebotenen Dienstes und der vom Dienstnutzer erwarteten Schnittstelle.

Die Verwendung einer solchen Subtyp–Relation ermöglicht die Zusammenarbeit von Dienstnutzern und Dienstanbietern, die keine A–priori–Kenntnisse in Bezug auf die *Schnittstelle* des Dienstes haben. Alleine durch die Typkonformität der angebotenen und der vom Dienstnutzer erwarteten Schnittstelle ist jedoch nicht sichergestellt, daß die *Semantik* des angebotenen Dienstes der vom Dienstnutzer erwarteten entspricht. Zur Lösung dieses Problems wurde versucht, Subtyping auf die Semantik von Diensten auszudehnen [29]. Es hat sich jedoch gezeigt, daß man dazu die semantischen Typ–Sprachen, auf denen die Subtyp–Relation definiert wird, stark einschränken muß, um die Berechnung der Typkonformität nicht $\mathcal{NP}$–hart werden zu lassen [46]. Ein anderer vielversprechender Ansatz ist die Verwendung *wissensbasierter Methoden* aus dem Bereich der KI zur Definition einer semantischen Subtyp–Relation [47].

Die Problematik der semantischen Konformität zwischen Erwartungshaltung des Dienstnutzers und angebotenen Dienst ist jedoch nicht Gegenstand dieser Arbeit. Hier wird lediglich ein Weg aufgezeigt, wie Dienstnutzer transparent auf Dienste zugreifen können, deren Schnittstellen über eine Subtyp–Relation in Beziehung zur vom Dienstnutzer erwarteten Schnittstelle stehen. Dazu wird ein Typsystem vorgestellt, das mächtig genug ist, um das durch die CORBA–IDL definierte Typsystem darauf abbilden zu können.

6.3.1 Typsystem

Das hier vorgestellte rekursive Typsystem ist eine Erweiterung des in [32] diskutierten derart, daß eine Abbildung der CORBA–IDL auf das erweiterte Typsystem möglich ist. Ebenso wie in [32] wird zunächst eine Typ–Sprache *IDL–Type* definiert, auf deren Basis dann Subtyping–Regeln vorgestellt werden.

Notationen

Im Rest dieses Abschnittes werden folgende Notationen verwendet:

- $r, s, t, \ldots$ bezeichnen Typvariablen und –konstanten; die Menge aller Typvariablen und Konstanten ist $TVar$.
- $\alpha, \beta, \gamma, \ldots$ bezeichnen Typen.
- $INST(\alpha)$ bezeichnet die Menge aller Instanzen (Werte) eines Typs α.
- $A, B, C, \ldots$ bezeichnen Instanzen eines Typs, d.h. $A \in INST(\alpha)$, $B \in INST(\beta)$, $C \in INST(\gamma)$.
- $a, b, c, \ldots$ bezeichnen Label.
- $i, j, k, \ldots$ bezeichnen nichtnegative ganze Zahlen.
- $\alpha\,[\beta/t]$ bedeutet die Ersetzung von t durch β in α.
- $\alpha \preceq \beta$ bedeutet α ist Subtyp von β.

Typ–Sprache *IDL–Type*

Die Sprache *IDL–Type* ist eine Erweiterung der in [32] vorgestellten Sprache *Type* um die Konstrukte der CORBA–IDL:

$$
\begin{array}{rcl}
\alpha & ::= & t \\
& | & \top \\
& | & \bot \\
& | & \text{short} \\
& | & \text{ushort} \\
& | & \text{long} \\
& | & \text{ulong} \\
& | & \text{float} \\
& | & \text{double} \\
& | & \text{array}(\alpha, i) \\
& | & \text{sequence}(\alpha, i) \\
& | & \text{enum}(a_1, \ldots, a_n) \\
& | & \text{struct}(a_1 : \alpha_1, \ldots, a_n : \alpha_n) \\
& | & \text{union}(\gamma, (C_1 : a_1 : \alpha_1, \ldots, C_n : a_n : \alpha_n)) \\
& | & \alpha_1 \times \ldots \times \alpha_n \\
& | & \alpha \longrightarrow \beta \\
& | & \mu t.\alpha
\end{array}
$$

Die speziellen Typen $\bot$ und $\top$ nehmen die Rollen des kleinsten und des größten Typs ein, d.h. $\bot$ ist Subtyp von jedem Typ und jeder Typ ist Subtyp von $\top$. Der Übersichtlichkeit halber sind alle IDL–Datentypen, die ausschließlich mit $\top$ und $\bot$ in einer Subtyp–Beziehung stehen, nicht Teil der Sprache *IDL–Type*. $\bot$ entspricht `void` in der CORBA–IDL, eine direkte Entsprechung für $\top$ gibt es in der CORBA–IDL nicht.

Sequenzen und Arrays sind durch einen Elementtyp und eine Index–Grenze parameterisiert. Im Falle von Sequenzen ist die Index–Grenze die maximale Größe, unbeschränkte Sequenzen haben eine Index–Grenze von 0. Zur Konstruktion von mehrdimensionalen Arrays werden einfach eindimensionale Arrays geschachtelt. Beispielsweise entpricht das Array

```
// IDL
short arr[3][2];
```

in der Sprache *IDL–Type* dem Ausdruck

$$\mathrm{array}(\mathrm{array}(\mathrm{short}, 2), 3)$$

Es gibt keine spezielle Regel für Exceptions, weil diese sich für die Belange eines Typsystems nicht von Strukturen unterscheiden.

Diskriminierte Unions verfügen über einen Diskriminatortyp γ. Jedes Element der Union besteht aus einem Tripel, das einen Wert C_i enthält, der eine Instanz des Disriminatortyps ist.

$\alpha \longrightarrow \beta$ bezeichnet eine Operation mit Eingabetyp α und Ausgabetyp β. Um Operationen mit mehreren Ein– bzw. Ausgabeparametern zu modellieren, wird die Regel für das kartesische Produkt verwendet. Beispielsweise entspricht die Operation

```
// IDL
short op (in long x, out float y, inout double z);
```

in der Sprache *IDL–Type* dem Ausdruck

$$\mathrm{long} \times \mathrm{double} \longrightarrow \mathrm{short} \times \mathrm{float} \times \mathrm{double}$$

Die letzte Regel dient zur Definition rekursiver Typen, indem $\mu t.\alpha$ die frei in α vorkommende Typvariable t bindet. Die CORBA–IDL schränkt die Konstruktion rekursiver Typen ein, so daß t in α ausschließlich als Elementtyp einer Sequenz verwendet werden darf. Beispielsweise wird die in CORBA–IDL zulässige Definition der folgenden rekursiven Struktur:

```
// IDL
struct S {
    long val;
    sequence<S> rec;
};
```

in der Sprache *IDL–Type* durch folgenden Ausdruck repräsentiert:

$$\mu t.\text{struct}(\text{val} : \text{long}, \text{rec} : \text{sequence}(t, 0))$$

Konstruiert man aus CORBA–IDL–Datentypen entsprechende Ausdrücke in der Sprache *IDL–Type*, so kommen darin keine freien Variablen vor. Typvariablen werden also im Zusammenhang mit CORBA–IDL nur zur Konstruktion von rekursiven Typen verwendet.

Typgleichheit

Gleichheit von Typen definiert eine Äquivalenzrelation. Einfache Datentypen gleichen nur sich selbst; für zusammengesetzte Typen ist Gleichheit über die Gleichheit der Elementypen definiert, wobei die Werte von Labeln nicht mit in Betracht gezogen werden. Die genaue Definition von Typgleichheit in *IDL–Type* ist analog zu der in [32] für *Type* angegebenen und wird deshalb hier nicht aufgeführt.

Subyping

Informal kann die Aussage „A ist Subtyp von B“ durch das *Prinzip der Ersetzbarkeit* erklärt werden: Überall, wo ein Objekt vom Typ B erwartet wird, kann ohne Schaden auch ein Objekt vom Typ A verwendet werden.

Formal wird die Subtyp–Relation durch eine Menge von Schlußregeln der Form

$$\frac{\Gamma \vdash \alpha_1 \preceq \beta_1, \Gamma \vdash \alpha_2 \preceq \beta_2}{\Gamma \vdash \alpha \preceq \beta}$$

definiert. Diese Art der Darstellung bedeutet, daß man um $\Gamma \vdash \alpha \preceq \beta$ zu zeigen, erst $\Gamma \vdash \alpha_1 \preceq \beta_1$ und $\Gamma \vdash \alpha_2 \preceq \beta_2$ zeigen muß. Um nachzuweisen, daß zwei Typausdrücke in einer Subtyp–Relation stehen, muß man informal die jeweils passenden Regeln rückwärts anwenden. Während der Anwendung dieser Regeln wird Γ mit Relationen der Form $\alpha \preceq \beta$ „gefüllt“, d.h. Γ repräsentiert schon berechnete Subtyp–Beziehungen zwischen Teilen der zu untersuchenden Typausdrücke. Formal gilt $\alpha \preceq \beta$, wenn man $\emptyset \vdash \alpha \preceq \beta$ herleiten kann. Hier nun der Wortlaut der Subtyping–Regeln:

$$\frac{\alpha = \beta}{\Gamma \vdash \alpha \preceq \beta} \quad (6.1)$$

$$\frac{\Gamma \vdash \alpha \preceq \beta, \Gamma \vdash \beta \preceq \gamma}{\Gamma \vdash \alpha \preceq \gamma} \quad (6.2)$$

$$\frac{t \preceq s \in \Gamma}{\Gamma \vdash t \preceq s} \quad (6.3)$$

$$\frac{}{\Gamma \vdash \bot \preceq \alpha} \quad (6.4)$$

$$\frac{}{\Gamma \vdash \alpha \preceq \top} \quad (6.5)$$

$$\frac{}{\Gamma \vdash \text{long} \preceq \text{double}} \quad (6.6)$$

$$\frac{}{\Gamma \vdash \text{float} \preceq \text{double}} \quad (6.7)$$

$$\frac{}{\Gamma \vdash \text{ulong} \preceq \text{double}} \quad (6.8)$$

$$\frac{}{\Gamma \vdash \text{short} \preceq \text{long}} \quad (6.9)$$

$$\frac{}{\Gamma \vdash \text{ushort} \preceq \text{long}} \quad (6.10)$$

$$\frac{}{\Gamma \vdash \text{short} \preceq \text{float}} \quad (6.11)$$

$$\frac{}{\Gamma \vdash \text{ushort} \preceq \text{float}} \quad (6.12)$$

$$\frac{}{\Gamma \vdash \text{ushort} \preceq \text{ulong}} \quad (6.13)$$

$$\frac{\Gamma \vdash \alpha_2 \preceq \alpha_1, \Gamma \vdash \beta_1 \preceq \beta_2}{\Gamma \vdash \alpha_1 \longrightarrow \beta_1 \preceq \alpha_2 \longrightarrow \beta_2} \quad (6.14)$$

$$\frac{i = j, \Gamma \vdash \alpha \preceq \beta}{\Gamma \vdash \text{array}(\alpha, i) \preceq \text{array}(\beta, j)} \quad (6.15)$$

$$\frac{i = j, \Gamma \vdash \alpha \preceq \beta}{\Gamma \vdash \text{sequence}(\alpha, i) \preceq \text{sequence}(\beta, j)} \quad (6.16)$$

$$\frac{n \leq m, \forall i \in \{1, \ldots, n\} : \Gamma \vdash \alpha_i \preceq \beta_i}{\Gamma \vdash \text{struct}(a_i : \alpha_1, \ldots, a_n : \alpha_n) \preceq \text{struct}(b_i : \beta_1, \ldots, b_n : \beta_n)} \quad (6.17)$$

$$\frac{n \leq m, \forall i \in \{1, \ldots, n\} : a_i = b_i}{\Gamma \vdash \text{enum}(a_i, \ldots, a_n) \preceq \text{enum}(b_i, \ldots, b_n)} \quad (6.18)$$

$$\frac{n \leq m, \Gamma \vdash \gamma \preceq \delta, \forall i \in \{1, \ldots, n\} : \alpha_i \preceq \beta_i \wedge A_i = B_i}{\Gamma \vdash \begin{array}{l}\text{union}(\gamma, (A_i : a_i : \alpha_i, \ldots, A_n : a_n : \alpha_n)) \preceq \\ \text{union}(\delta, (B_i : b_i : \beta_i, \ldots, B_n : b_n : \beta_n))\end{array}} \quad (6.19)$$

$$\frac{\forall i \in \{1, \ldots, n\} : \Gamma \vdash \alpha_i \preceq \beta_i}{\Gamma \vdash \alpha_1 \times \ldots \times \alpha_n \preceq \beta_1 \times \ldots \times \beta_n} \quad (6.20)$$

$$\frac{\Gamma \cup \{t \preceq s\} \vdash \alpha \preceq \beta}{\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta} \quad (6.21)$$

Regel 6.1 besagt, daß gleiche Typen auch Subtypen sind. Regel 6.2 macht die Subtyp–Beziehung zu einer transitiven Relation. Regel 6.3 ist eine Hilfsregel für die Bestimmung von Subtyp–Beziehungen zwischen rekursiven Typen. Die Regeln

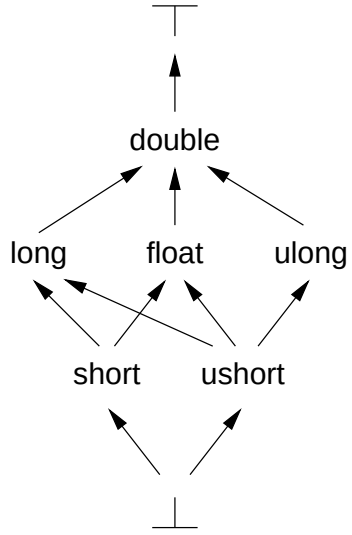


Abbildung 6.2: Subtyp–Beziehungen zwischen einfachen Datentypen

6.4 und 6.5 besagen, daß $\perp$ Subtyp von jedem Typ ist und jeder Typ Subtyp von $\top$ ist. Regeln 6.6 bis 6.13 geben die Subtypbeziehungen für die einfachen Datentypen wieder. Generell ist ein einfacher Datentyp α Subtyp eines einfachen Datentyps β, wenn $INST(\alpha) \subset INST(\beta)$. Der besseren Anschaulichkeit wegen sind die Subtypbeziehungen zwischen den einfachen Datentypen nochmals graphisch in Abbildung 6.2 dargestellt.

Regel 6.14 sagt aus, daß zwei Operationen in einer Subtyp–Relation stehen, wenn die Eingabeparameter co– und die Ausgabeparameter kontravariant sind. Arrays und Sequenzen stehen zueinander in einer Subtyp–Relation, wenn die Elementypen in einer Subtyp–Beziehung stehen und die Index–Grenzen gleich sind (Regeln 6.15 und 6.16). Eine Struktur ist Subtyp einer zweiten, wenn die Elemente der ersten Subtypen der entsprechenden Elemente der zweiten Struktur sind. Dabei kann die zweite Struktur durchaus mehr Elemente als die erste haben (Regel 6.17). Für Aufzählungstypen (Regel 6.18) gilt ähnliches; hier müssen aber die Label übereinstimmen. Eine Union ist Subtyp einer zweiten (Regel 6.19), wenn der Diskriminator–Typ der ersten Subtyp des Diskriminator–Typs der zweiten ist. Außerdem müssen die Elemente der ersten Union Subtypen der zugeordneten Elemente der zweiten sein und die jeweiligen Label müssen übereinstimmen. Man beachte, daß die Label durchaus verschiedenen Typs sein und trotzdem gleiche Werte aufweisen können. Beispielsweise ist die Zahl 3 sowohl in $INST$(short) als auch in $INST$(long). Nach Regel 6.20 ist die Subtyp–Beziehung zwischen zwei kartesichen Produkten über Subtyp–Beziehungen zwischen den Elementen definiert. Regel 6.21 gibt die Subtyp–Beziehung zwischen rekursiven Typen wieder. Dabei darf die Typvariable t bzw. s nicht in β bzw. α vorkommen. Um zu zeigen, daß ein rekursiver Typ Subtyp eines anderen ist, muß man zeigen, daß die „Körper“ der rekursiven Typen in einer Subtyp–Beziehung stehen, wobei man

für die durch den μ–Operator gebundenen Typvariablen jeweils wieder die Definition einsetzt. Um ein unendliches Einsetzen zu verhindern, wird vorher Γ um $t \preceq s$ erweitert, was zusammen mit Regel 6.3 die Terminierung des Verfahrens garantiert.

Typanpassung

Wie im vorhergehenden Abschnitt bereits angeführt, kann man die Subtyp–Relation $\alpha \preceq \beta$ so interpretieren, daß überall dort, wo Instanzen des Typs β erwartet werden, ohne Schaden Instanzen des Typs α verwendet werden können. Das erfordert jedoch die Umwandlung von Instanzen des Typs α in Instanzen des Typs β. Diese Umwandlung bezeichnet man mit *coercion* oder *Typanpassung*.

6.3.2 Beispiel

Eine Schnittstelle, die ein Bankkonto modelliert, verfügt unter anderem über eine Operation zum Einzahlen eines Geldbetrages:

```
// IDL

// vom Klient erwartete Schnittstelle
void einzahlen_klient (in long betrag);

// vom Dienst bereitgestellte Schnittstelle
void einzahlen_server (in double betrag);
```

Die vom Klient erwartete und die vom Dienst bereitgestellte Schnittstelle unterscheiden sich im Typ des Arguments. Übersetzt in die Sprache *IDL–Type* ergeben sich folgende Typausdrücke:

$$\texttt{einzahlen\_klient}: \quad \text{long} \longrightarrow \bot$$

$$\texttt{einzahlen\_server}: \quad \text{double} \longrightarrow \bot$$

Durch Anwendung der Regeln 6.14, 6.6 und 6.1 in dieser Reihenfolge läßt sich nachweisen, daß `einzahlen_server` $\preceq$ `einzahlen_klient`. Somit kann nach dem Prinzip der Ersetzbarkeit `einzahlen_server` überall dort verwendet werden, wo `einzahlen_klient` erwartet wird. Durch Typanpassung des Arguments `betrag` kann der Klient also den bereitgestellten Dienst benutzen.

6.4 Design des MICO–DII

Das *Dynamic Invocation Interface (DII)* spielt in MICO eine zentrale Rolle, weil es keinen besonderen statischen Aufruf–Adapter gibt und die Stub–Objekte somit

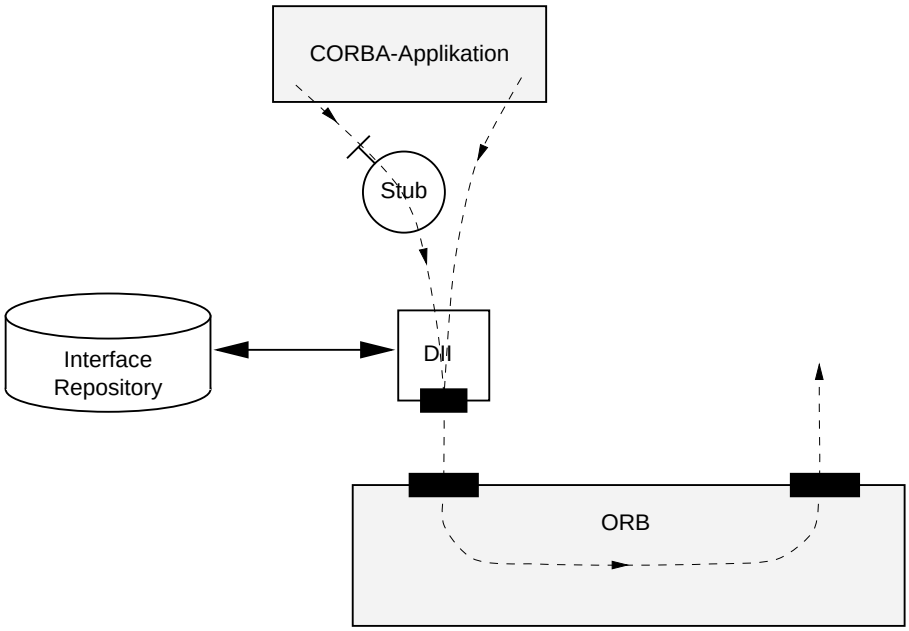


Abbildung 6.3: Einbindung des DII

auch das DII verwenden. Auf Basis der in Kapitel 3 vorgestellten Mikrokern–Architektur ist es aber jederzeit möglich, weitere Aufruf–Adapter bereitzustellen. Abbildung 6.3 zeigt die Einbindung des DII als einen Service außerhalb des Mikrokern–ORB. Zur Unterstützung von Subtyping und zur Typüberprüfung arbeitet das DII eng mit dem Interface Repository zusammen.

Kern des DII bilden `Any`–Objekte zur Repräsentation beliebiger IDL–Datentypen. Die Schnittstelle dieser Objekte verfügt zwar über Operationen zum Einfügen und Auslesen einfacher Datentypen, entsprechende Operationen für zusammengesetzte Datentypen wie Strukturen und Unions sind aber nicht spezifiziert.

Bis auf diese fehlende Schnittstelle ist die Spezifikation des DII so detailliert, daß für eine Implementierung kein großer Spielraum bleibt. Daher wird im folgenden nur näher auf den Entwurf einer Schnittstelle zum Einfügen und Auslesen zusammengesetzer Datentypen in bzw. aus `Any`–Objekten und auf die Unterstützung von Subtyping gemäß dem in Abschnitt 6.3 vorgestellten Modell eingegangen.

6.4.1 Zusammengesetzte Datentypen

Wie bereits in Abschnitt 6.1.1 beschrieben, sind die Operationen zum Einfügen und Auslesen von einfachen Datentypen in bzw. aus `Any`–Objekten typsicher, was den Umgang mit `Any`–Objekten wenig anfällig für Fehler seitens des Programmierers macht.

Es wäre wünschenswert, die Operationen für zusammengesetzte Datentypen ebenfalls typsicher zu machen.

Leider ist es beim Einfügen eines zusammengesetzten Datums prinzipiell unmöglich, wie bei einfachen Datentypen alleine aus den aufgerufenen Einfüge–Operationen auf den Typ des eingefügten Datums zu schließen. Als Beispiel betrachte man diskriminierte Unions: Beim Einfügen wird nur der Wert des Diskriminators und der Wert des zugehörigen „Case“ betrachtet, der Typ der Union enthält aber Informationen über *alle* „Cases“.

Daher muß der Programmierer vor dem Einfügen eines zusammensetzten Datentypes zunächst den Typ in Form eines `TypeCode` vorgeben. Beim Einfügen der Daten überprüft `Any` dann, ob die eingefügten Werte konform zum vorgegebenen Typ sind.

Die `Any`–Schnittstelle wurde durch Vererbung um folgende Operationen erweitert (in Auszügen):

```
// IDL
interface MICOAny : Any {
    // Einfuege-Operationen
    void type (in TypeCode type);

    boolean struct_put_begin ();
    boolean struct_put_end ();

    boolean union_put_begin ();
    boolean union_put_selection (in long index);
    boolean union_put_end ();

    boolean any_put (in any a);
    ...

    // Auslese-Operationen
    boolean struct_get_begin ();
    boolean struct_get_end ();

    boolean union_get_begin ();
    boolean union_get_selection (in long index);
    boolean union_get_end ();

    boolean any_get (out any a);
    ...
};
```

`type()` dient zum Vorgeben des Typs vor dem Einfügen. Für jede zusammengesetzte Datenstruktur existieren Operationen, die den Beginn und das Ende der

Datenstruktur anzeigen. Damit sieht das Einfügen einer Struktur folgendermaßen aus (Auslesen analog, jedoch ohne vorherigen Aufruf von `type()`):

```
// Pseudo-Kode
MICOAny mico_any;

mico_any.type (<TypeCode fuer Struktur>);
mico_any.struct_put_begin ();
// Elemente der Struktur einfuegen
mico_any.struct_put_end ();
```

Das Einfügen bzw. Auslesen geschieht je nach Typ der Elemente durch die Operationen für einfache Datentypen oder durch geschachtelte Anwendung der Operationen für zusammengesetzte Datentypen.

Zusätzlich werden Operationen bereitgestellt, um den in einem `Any` enthaltenen Wert in ein anderes `Any` einzufügen bzw. auszulesen. Diese Operationen verhalten sich so, als ob man von Hand die Daten aus dem einen `Any` auslesen und sofort wieder in das andere einfügen würde. Wie sich im nächsten Abschnitt zeigen wird, sind diese Operationen die Grundlage für die Typanpassung.

Trotz ihrer Nützlichkeit bewirkt die direkte[2] Verwendung von `MICOAny` in CORBA–Applikationen den Verlust von Portabilität zu anderen CORBA–Produkten, die nicht über diese Schnittstelle verfügen. Daher hat die OMG in [36] zur Spezifikation einer solchen Schnittstelle aufgerufen. Die als Antwort darauf unter dem Namen *Dynamic Any* [40] vorgeschlagene Spezifikation ist keine Erweiterung der `Any`–Schnittstelle, sondern definiert eine davon unabhängige Menge von IDL–Schnittstellen.

Neben dem Einfügen und Auslesen von zusammengesetzten Datentypen in bzw. aus `Any`–Objekten bietet Dynamic Any eine umfangreiche Funktionalität zur Traversierung und Manipulation der enthaltenen zusammengesetzten Datentypen. Bei der im Rahmen dieser Diplomarbeit vorgenommenen Referenzimplementierung dieser Schnittstelle auf Basis von `MICOAny` hat es sich gezeigt, daß sich diese umfangreiche Funktionalität nachteilig auf die Effizienz auswirkt.

Daher ist Dynamic Any kein Ersatz für `MICOAny`, sondern eine Ergänzung, die die Funktionalität von `MICOAny` über eine standardisierte, aber weniger effiziente Schnittstelle zur Verfügung stellt.

6.4.2 Subtyping

Das MICO–DII unterstützt Subtyping auf Basis des in Abschnitt 6.3 vorgestellten Modells mit der Einschränkung, daß die Methodennamen der vom Klient erwarteten und der vom Dienst bereitgestellten Schnittstellen gleich sein müssen.

[2] Die indirekte Verwendung über Stub–Objekte birgt diese Probleme nicht.

Neben der Typüberprüfung auf Schnittstellenebene muß nämlich mittels geeigneter Dienstvermittlungs–Mechanismen sichergestellt werden, daß die Semantik einer von einem Dienst bereitgestellten Schnittstelle den Erwartungen des Klienten gerecht wird. Diese Vermittlung geschieht aber aus Effizienzgründen meist nicht auf Methodenebene, sondern auf Objektebene. Für das DII stellt sich damit die Frage, wie Methoden der vom Klienten erwarteten Schnittstelle eindeutig zu Methoden der vom Dienst tatsächlich bereitgestellten Schnittstelle zugeordnet werden können. Die oben genannte Beschränkung resultiert daraus, daß das MICO–DII die Zuordnung über die Namen der Methoden vornimmt.

Da vom Dienstvermittler bereits während des Vermittlungsvorgangs festgestellt wurde, daß die Methoden der vom Dienst bereitgestellten Schnittstelle Subtypen der gleichnamigen Methoden der vom Klienten erwarteten Schnittstelle sind, beschränkt sich die Aufgabe des DII darauf, gegebenfalls vor dem Methodenaufruf eine Typanpassung der Eingabeparameter und nach dem Methodenaufruf eine Typanpassung der Ausgabeparameter vorzunehmen.

Grundlage für die Typanpassung bildet das im vorhergehenden Abschnitt beschriebene `MICOAny`. Dessen Operationen zum Auslesen von einfachen und zusammengesetzten Datentypen sind alle so realisiert, daß ein Auslesen auch dann möglich ist, wenn der im `MICOAny` enthaltene Typ Subtyp vom ausgelesenen Typ ist. Die Typanpassung wird dann automatisch beim Auslesen vorgenommen. Beispielsweise ist folgendes möglich:

```
// Pseudo-Kode
MICOAny any;
any.put_long (3);

double d;
any.get_double (d);
```

Beim `get_double()` nimmt `MICOAny` automatisch eine Typanpassung von `long` nach `double` vor. Ähnliches gilt für die Auslese–Operationen zusammengesetzter Datentypen.

Das DII führt die Typanpassung für jeden Parameter der augerufenen Methode folgendermaßen durch:

```
// Pseudo-Kode
MICOAny klient_param = ...
TypeCode server_param_type = ...

MICOAny server_param;
server_param.type (server_param_type);
server_param.any_put (klient_param);
```

Der vom Aufrufer übergebene Parameter liegt in Form von `klient_param` vor. Das DII erfragt nun beim Interface Repository den Typ des entsprechenden Parameters in der Signatur der vom Dienst angebotenen Methode und initialisiert `server_param` mit diesem Typ. `put_any()` verhält sich nun so, als würde man das in `klient_param` enthaltene Datum „von Hand“ auslesen und in `server_param` einfügen. Beim Auslesen nimmt `MICOAny` die Typanpassung auf den vom Dienst erwarteten Typ vor, so daß `server_param` danach den angepaßten Parameter enthält.

Da die Anfragen beim Interface Repository einen erheblich Aufwand bedeuten, kann man daß DII auch in einen Modus umschalten, in dem keine Typanpassung vorgenommen wird. Wird in diesem Modus eine Typanpassung erforderlich, so führt das zu einem Laufzeitfehler in Form einer Exception.

6.5 Zusammenfassung, Bewertung und Alternativen

Um CORBA an die Anforderungen offener Dienstmärkte anzupassen, wird die Verwendung von rekursivem Subtyping vorgeschlagen.

In MICO verwenden Stub–Objekte keinen besonderen statischen Aufruf–Adapter, sondern das DII. Dazu war es notwendig, `Any` um eine Schnittstelle zur Behandlung von zusammengesetzten Datentypen zu erweitern, da eine solche von CORBA nicht spezifiziert wird. Auf Basis dieser Schnittstelle wird Unterstützung für Subtyping ins DII integriert. Dieser Ansatz weist folgende Eigenschaften auf:

+ Flexibilität durch rekursives Subtyping
+ Unterstützung für zusammengesetzte Datentypen
− schlechte Performanz

Die schlechte Performanz resultiert daraus, daß für Stub–Objekte das DII anstelle eines besonders optimierten statischen Aufruf–Adapters verwendet wird (siehe dazu auch [16]). Dieser Weg wurde gewählt, weil Geschwindigkeit kein primäres Ziel dieser Arbeit ist. Durch Entwicklung eines statischen Aufruf–Adapters läßt sich dieser Nachteil aber beheben.

Eine weitere Performanz–Einschränkung resultiert aus den für die Subtyping–Unterstützung notwendigen Zugriffen auf das Interface Repository. Durch Einsatz eines Caches [31] ließe sich der Zugriff erheblich beschleunigen. Die Verwendung des Interface Repository liese sich ganz vermeiden, wenn es in der Aufrufkette eine Stelle gäbe, an der sowohl die Signatur der vom Klient erwarteten Methode als auch der vom Dienst bereitgestellten Methode implizit bekannt wären. Leider gibt es eine solche Stelle nicht, da aus Effizienzgründen keine Typinformationen

zwischen Klient und Dienst übertragen werden (zumindest bei Verwendung von GIOP).

Kapitel 7

Zusammenfassung und Ausblick

Um in verschiedenen Anwendungsumgebungen eingesetzt werden zu können, läßt die CORBA–Spezifikation einen weiten Spielraum für Implementierungen. Sollte CORBA in einem speziellen Umfeld eingesetzt werden, so war bisher eine Neu–Implementierung notwendig, da herkömmliche CORBA–Implementierungen nicht oder nur sehr eingeschränkt an spezielle Anwendungsumgebungen anpaßbar sind. In dieser Arbeit wurde ein Ansatz für eine *erweiterbare* CORBA–Implementierung vorgestellt und implementiert.

Dazu wurde der Mikrokern–Ansatz aus dem Bereich der Betriebssysteme auf CORBA übertragen. Die Anpassung an ein neues Einsatzgebiet erfolgt durch Bereitstellung neuer *Services*, die jederzeit (auch bei laufendem Betrieb) an den Mikrokern–ORB gebunden und wieder entfernt werden können. Auf Basis dieser Mikrokern–Architektur wurde eine zur Version 2.0 der CORBA–Spezifikation konforme CORBA–Implementierung entworfen und realisiert.

Einen Schwerpunkt bildete dabei die Bereitstellung von Mechanismen, um den Anforderungen *offener Dienstemärkte* gerecht zu werden, die sich dadurch auszeichnen, daß Dienstanbieter keine A–priori–Kenntnisse über Dienstnutzer und deren Schnittstellen (und umgekehrt) haben.

Proof of Concept

Es bleibt zu zeigen, ob die in dieser Arbeit entwickelten Konzepte und deren Implementierung in der Praxis die gestellte Forderung nach Erweiterbarkeit erfüllen. Anzeichen dafür geben einige Projekte, bei denen MICO inzwischen eingesetzt wird.

Im Rahmen des Projektes MAQS [5] werden Konzepte zur Integration von *Dienstgüte–Aspekten* in CORBA entwickelt. Der Schwerpunkt liegt dabei darauf, einen allgemeinen Rahmen zur Unterstützung *beliebiger* Dienst–Qualitäten zu schaffen. Dazu ist eine weitreichende Unterstützung durch den ORB erforderlich, die auf Basis von MICO prototypisch realisiert werden soll. Dabei wird sich zeigen,

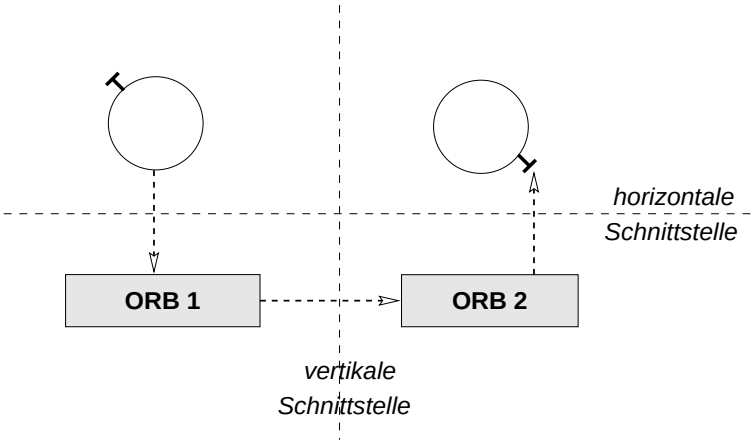


Abbildung 7.1: Vertikale und horizontale Schnittstellen

ob die MICO zugrundeliegenden Konzepte ihr Ziel erreichen.

Eine Folge–Diplomarbeit soll anhand des *Portable Object Adapter (POA)* [40] zeigen, ob die Erweiterung des Mikrokern–ORB um neue Objekt–Adapter praktikabel ist.

Erfahrungen mit der CORBA–Spezifikation

Die Umsetzung der CORBA–Spezifikation in eine Implementierung erfordert einen erheblichen konzeptionellen und technischen Aufwand. Ursache dafür ist, daß nur die vom Anwender sichtbaren Schnittstellen des CORBA–Systems spezifiziert werden. Ziel dieser Vorgehensweise ist die Sicherstellung von Portabilität zwischen verschiedenen CORBA–Implementierungen, ohne das Anwendungsfeld von CORBA zu stark einzuschränken.

Die Qualität der CORBA–Spezifikation schwankt insbesondere im Hinblick auf Portabilität stark. Während von Klienten benötigte Schnittstellen hinreichend detailliert definiert sind, läßt die Spezifikation der von Objekt–Implementierungen benötigten Schnittstellen teilweise stark zu wünschen übrig. Die Probleme, an denen die OMG teilweise bereits arbeitet, gliedern sich in drei Kategorieren:

- fehlende Funktionalität
- nicht spezifizierte Schnittstellen
- unklare Semantik von spezifizierten Schnittstellen

Kapitel 0.6 der CORBA–Spezifikation definiert unter dem Begriff *CORBA–Konformität* die Bedingungen, unter denen eine CORBA–Implementierung der Spezifikation entspricht. Erstaunlicherweise zeigte eine im Rahmen dieser Arbeit

vorgenommene Analyse, daß neben MICO kaum eine der existierenden CORBA–Implementierungen, insbesondere auch der kommerziell verfügbaren, konform zur Version 2 der CORBA–Spezifikation ist.

Hier wird der Badarf nach einem offiziellen Konformitäts–Test für CORBA–Implementierungen deutlich. Wie in Abbildung 7.1 dargestellt verfügt ein CORBA–System über zwei Arten von Schnittstellen, die relevant für einen solchen Test sind: *vertikale* Schnittstellen zwischen CORBA–Produkten verschiedener Hersteller und *horizontale* Schnittstellen zwischen CORBA–Produkt und Programmierer. Testumgebungen für vertikale Schnittstellen existieren bereits (beispielsweise [42]). Automatisierte Tests der Konformität horizontaler Schnittstellen existieren noch nicht. Eine mögliche Ursache dafür sind die oben angedeuteten Unzulänglichkeiten der CORBA–Spezifikation.

Auswirkungen der Arbeit

Im Mai 1997 wurde MICO in einer ersten Version der breiten Öffentlichkeit zugänglich gemacht. Seitdem hat es sich gezeigt, daß international ein reges Interesse an MICO besteht. Als Diskussionsforum wurde daher eine Mailing–Liste eingerichtet, auf der inzwischen über 200 Anwender und Entwickler die Verbesserung und Weiterentwicklung von MICO diskutieren. Darüber hinaus wurden Teile der aktuellen MICO–Version 2.0.3 (Object Services, Anbindungen an diverse Programmiersprachen) von Dritten beigesteuert.

Die Einbeziehung einer Vielzahl von Anwendern und Entwicklern in die über das Internet koordinierte kollaborative Entwicklung von MICO ist sowohl für Anwender als auch Entwickler von Vorteil. Anwender profitieren von der schnellen Beantwortung von Fragen und Behebung von Fehlern durch den gemeinsamen Erfahrungsschatz der Anwender und Entwickler. Entwickler werden von der Beantwortung häufig gestellter Fragen entlastet und erfahren direkt von Fehlern in der Software, sowie Problemen und Wünschen der Anwender.

Abschließend seien noch einige Aktivitäten genannt, die auf der vorliegenden Arbeit basieren:

- An der Professur „Verteilte Systeme und Betriebssysteme“ wurde im Wintersemester 1997/98 ein Praktikum durchgeführt, bei dem Studenten anhand von MICO der interne Aufbau einer CORBA–Implementierung nahegebracht wurde.
- Im Rahmen des KDE–Projektes [27] wird untersucht, wie auf Basis von MICO eine Infrastruktur für zusammengesetzte verteilte Dokumente entworfen und realisiert werden kann.

Die Ergebnisse dieser Arbeit sind im Rahmen einer Veröffentlichung beim dpunkt–Verlag in Zusammenarbeit mit Morgan Kaufmann Publishers, Inc. erschienen [49].

Literaturverzeichnis

[1] Abelson, H., G. J. Sussman und J. Sussman: *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Active Group: *http://www.activex.org/.*

[3] Amadio, R.M. und L. Cardelli: *Subtyping Recursive Types.* ACM Transactions on Programming Languages and Systems, 15(4):575–631, September 1993.

[4] Bapat, S.: *Object–Oriented Networks, Models for Architecture, Operations, and Management.* Prentice/Hall International, 1994.

[5] Becker, C. und K. Geihs: *MAQS – Generic QoS Management for Adaptive Applications.* In: *IEEE Workshop on Middleware for Distributed Real–Time Systems and Services*, San Francisco, USA, Dezember 1997.

[6] Berners-Lee, T., L. Masinter und M. McCahill: *RFC 1738: Uniform Resource Locators (URL)*, Dezember 1994.

[7] Booch, G.: *Object Oriented Design with Applications.* Benjamin Cummings Publishing Company, Inc, Redwood City, California, 1991.

[8] Brose, G.: *JacORB. http://www.inf.fu-berlin.de/~brose/jacorb/.*

[9] Burger, C.: *Groupware.* dpunkt – Verlag für digitiale Technologie GmbH, 1997.

[10] Component Integration Laboratories (CIL): *Shaping Tomorrow's Software (White Paper), ftp://cil.org/pub/cilabs/tech/opendoc/OD–overview.ps*, 1994.

[11] Day, J. D. und H. Zimmermann: *The OSI Reference Model.* In: *Proceedings of the IEEE*, Band 71, Dezember 1983.

[12] Dupuy, F., G. Nilsson und Y. Inoue: *The TINA Consortium: Toward Networking Telecommunications Information Services.* IEEE Communications Magazine, Seiten 78–83, November 1995.

[13] FAYAD, M. E. und D. C. SCHMIDT: *Object–Oriented Application Frameworks.* Communications of the Association for Computing Machinery, Oktober 1997.

[14] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Design Patterns: Elements of Reusable Object–Oriented Software.* Addison–Wesley Publishing Company, 1995.

[15] GEIHS, K. und U. HOLLBERG: *A Retrospective on DACNOS.* Communications of the Association for Computing Machinery, 33(4), 1990.

[16] GOKHALE, A. und D. C. SCHMIDT: *The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High–Speed ATM Networks.* In: *Proceedings of GLOBECOMM '96*, London, England, November 1996.

[17] GOSCINSKI, A.: *Distributed Operating Systems, The Logical Design.* Addison–Wesley Publishing Company, 1991.

[18] GRÜNDER, H.: *Zur Anwendung des Objekt–Modells in verteilten Systemen.* Doktorarbeit, J. W. Goethe–Universität Frankfurt/Main, 1998.

[19] HEWLETT–PACKARD COMPANY: *ORBplus. http://www.hp.com/.*

[20] IBM CORP.: *DSOM. http://www.ibm.com/.*

[21] IONA TECHNOLOGIES LTD.: *IONA's Object Database Adapter Framework. http://www-usa.iona.com/Press/PR/odaf.html.*

[22] IONA TECHNOLOGIES LTD.: *Orbix 2 — Distributed Object Technology.* Iona, 1996.

[23] *ITU.TS Recommendation X.901 — ISO/IEC 10746–1: Basic Reference Model of Open Distributed Processing Part 1: Overview and Guide to the use of the Reference Model*, Juli 1994.

[24] *ITU.TS Recommendation X.902 — ISO/IEC 10746–2: Basic Reference Model of Open Distributed Processing Part 2: Descriptive Model*, 1994.

[25] *ITU.TS Recommendation X.903 — ISO/IEC 10746–3: Basic Reference Model of Open Distributed Processing Part 3: Prescriptive Model*, Februar 1994.

[26] *ITU.TS Recommendation X.904 — ISO/IEC 10746–4: Basic Reference Model of Open Distributed Processing Part 4: Architectural Semantics*, 1994.

[27] KDE TEAM: *K Desktop Enviroment. http://www.kde.org/.*

[28] Landin, Peter: *A correspondence between Algol 60 and Church's lambda notation: Part I.* Communications of the Association for Computing Machinery, 1965.

[29] Liskov, B. und J. Wing: *Family Values: A Behavioral Notion of Subtyping.* Technischer BerichtCMU–CS–93–187, Computer Science Department, Carnegie Mellon University, Pittsburgh, Juli 1993.

[30] Maffeis, S.: *Adding Group Communication and Fault–Tolerance to CORBA.* In: *Proceedings of the USENIX Conference on Object–Oriented Technologies*, Juni 1995.

[31] Merle, P.: *CorbaScript.* In: *CORBA: Implementation, Use and Evaluation, ECOOP*, Jyväskylä, Finnland, Juni 1997.

[32] Najm, E. und J.B. Stefani: *A formal semantics for the ODP computational model.* Computer Networks and ISDN Systems, 27(8):1305–1329, Juli 1995.

[33] Netscape Comm. Corp.: *New Netscape ONE Platform Brings Distributed Objects to the Internet and Intranets. http://home.netscape.com/newsref/pr/newsrelease199.html*, Juli 1996.

[34] Object Management Group (OMG): *Compound Presentation and Compound Interchange Facilities*, 1995.

[35] Object Management Group (OMG): *CORBAservices: Common Object Services Specification*, März 1995.

[36] Object Management Group (OMG): *ORB Portability Enhancement RFP*, Juni 1995.

[37] Object Management Group (OMG): *Control and Management of A/V Streams RFP*, August 1996.

[38] Object Management Group (OMG): *Objects–by–value RFP*, Juni 1996.

[39] Object Management Group (OMG): *A Diskussion of the Object Management Architecture*, Januar 1997.

[40] Object Management Group (OMG): *ORB Portability Joint Submission*, Juni 1997.

[41] Object Management Group (OMG): *The Common Object Request Broker: Architecure and Specification, Revision 2.0*, Juli 1995.

[42] Object Management Group (OMG): *OMG Unveils CORBAnet Initiative. http://www.omg.org/pr96/corbanet.html*, Mai 1996.

[43] Object Oriented Concepts Inc.: *OmniBroker 2.0. http://www.ooc.com/.*

[44] Open Software Foundation: *Introduction to DCE.* Open Software Foundation, Inc., 1992.

[45] Oracle and Olivetti Reasearch Labs: *OmniORB 2.0. http://www.orl.co.uk/omniORB/.*

[46] Puder, A.: *Typsysteme für die Dienstvermittlung in offenen verteilten Systemen.* Doktorarbeit, J. W. Goethe–Universität Frankfurt/Main, 1997.

[47] Puder, A., S. Markwitz, F. Gudermann und K. Geihs: *AI–based Trading in Open Distributed Environments.* In: *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

[48] Puder, A. und K. Römer: *Using a Meta–Notation in a CORBA environment.* In: *CORBA: Implementation, Use and Evaluation, ECOOP*, Jyväskylä, Finnland, Juni 1997.

[49] Puder, A. und K. Römer: *MICO—MICO Is CORBA.* dpunkt – verlag für digitale Technologie GmbH, 1998.

[50] Rogerson, Dale: *Inside COM.* Microsoft Press, 1997.

[51] Römer, K. und A. Puder: *MICO.* http://www.vsb.cs.uni-frankfurt.de/~mico/.

[52] Rumbaugh, J.: *Object–Oriented Modeling and Design.* Prentice/Hall International, 1991.

[53] Schill, A.: *Remote Procedure Call: Fortgeschrittene Konzepte und Systeme – ein Überblick. Teil 2: Erweiterte RPC–Ansätze.* Informatik–Spektrum, Band 15, Heft 3, 1992.

[54] Schmidt, D. C.: *ACE: an Object–Oriented Framework for Developing Distributed Applications.* In: *Proceedings of the USENIX Conference on Object–Oriented Technologies.* USENIX Association, April 1994.

[55] Schmidt, D. C., D. L. Levine und S. Mungee: *The Design of the TAO Real–Time Object Request Broker.* Computer Communications Journal, 1997.

[56] SCHMIDT, D. C. und S. VINOSKI: *Object Adapters: Concepts and Terminology.* SIGS C++ Report Magazine, Oktober 1997.

[57] SIEGEL, J.: *CORBA: Fundamentals and Programming.* John Wiley & Sons Ltd., 1996.

[58] SOFTWARE AG: *EntireX DCOM Unix Port. http://www.sagus.com/prod–i~1/net–comp/dcom/index.html.*

[59] STROUSTRUP, B.: *Die C++ Programmiersprache.* Addison–Wesley Publishing Company, Second Auflage, 1992.

[60] SUNSOFT INC.: *Java RMI. http://java.sun.com/jdk/rmi/index.html.*

[61] TANENBAUM, A. S.: *Operating Systems – Design and Implementation.* Prentice/Hall International, 1987.

[62] TANENBAUM, A. S.: *Computer Networks.* Prentice/Hall International, second Auflage, 1989.

[63] TANENBAUM, A. S.: *Modern Operating Systems.* Prentice/Hall International, 1992.

[64] TRAN, F. D. und J. B. STEFANI: *Towards an extensible and modular ORB framework.* In: *CORBA: Implementation, Use and Evaluation, ECOOP*, Jyväskylä, Finnland, Juni 1997.

[65] VINOSKI, S.: *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments.* IEEE Communications Magazine, Februar 1997.

[66] VISIGENIC: *VisiBroker 3.0. http://www.visigenic.com/prod/.*

[67] WOOLDRIDGE, M. J. und N. R. JENNINGS (Herausgeber): *ECAI–94 Workshop on Agent Theories, Architectures, and Languages.* Springer Verlag, 1995.